



# apple machine language

Robert D. Rosen







# ***Apple Machine Language***



## **Apple Programming Series**

R.D. Rose, *Apple Machine Language*

A.B. Tucker, Jr., *Apple IIe: Programming in Basic*

A.B. Tucker, Jr., *Apple Basic: A Programming Guide*

A.B. Tucker, Jr., *Apple Pascal: A Programming Guide*



# ***Apple Machine Language***

**Robert D. Rosen**

**HOLT, RINEHART AND WINSTON**

|             |                |               |              |       |
|-------------|----------------|---------------|--------------|-------|
| New York    | Chicago        | San Francisco | Philadelphia |       |
| Montreal    | Toronto        | London        | Sydney       | Tokyo |
| Mexico City | Rio de Janeiro | Madrid        |              |       |

"Apple" is a registered trademark of Apple Computer, Inc. Unauthorized use of this trademark is contrary to the laws of the Federal Government.

This book is a tutorial guide to Apple Machine Language, not a formal specification of the software as delivered to the buyer now or in the future software revisions. Apple Computer, Inc. makes no warranties with respect to this book or to its accuracy in describing any version of Apple machine language.

Copyright © 1983 CBS College Publishing  
All rights reserved.  
Address correspondence to:  
383 Madison Avenue, New York, NY 10017

#### Library of Congress Cataloging in Publication Data

Rosen, Robert D.  
Apple machine language.

(Apple programming series)  
Includes index.

|                                |           |
|--------------------------------|-----------|
| I. Apple computer—Programming. | I. Title. |
| II. Series.                    |           |
| QA76.8.A66R67 1983 001.64'2    | 83-78     |
| ISBN 0-03-06336-2              |           |

Printed in the United States of America  
Published simultaneously in Canada

3 4 5 6 039 9 8 7 6 5 4 3 2 1

CBS COLLEGE PUBLISHING  
Holt, Rinehart and Winston  
The Dryden Press  
Saunders College Publishing



# ***Preface***

Although there are some excellent reference works on programming the Apple computer (6502 microprocessor) in machine language, most of them tend to be very difficult for the beginning programmer to understand. The purpose of this book is to provide an easy-to-read guide to Apple machine language programming.

It is assumed that the reader has a good background in BASIC programming. Such a background will provide the logical foundation for understanding the material presented here. Wherever possible, I have tried to relate machine language programming instructions to BASIC programming instructions. By explaining machine language in terms already familiar to the reader, it is hoped that learning machine language will be facilitated.

To program in machine language, it is necessary to have a very well-organized plan of attack. For this reason, the examples presented in this text are written in three steps. Beginning with the examples in Chapter 4, a flowchart, a rough assembly language version, and a finished listing are presented for each of the examples. The flowchart is really just a block diagram that outlines in simple terms the steps required to solve the given problem. To produce the rough assembly language version of the program, each block of the flowchart is translated directly into assembly language. Thus the reader will be able to determine precisely what each section of the rough assembly language program does simply by referring to the corresponding block of the flowchart. The finished listing is found by looking up (or else using a computer program that does this for you) the machine language codes that correspond to the assembly language shorthand given in the rough version of the program. It is suggested that the reader try to write some of the programs, referring to the text as little as possible. The documentation given in the text can be used for reference if questions arise.

The examples in the text were carefully chosen. It was my intention to present simple examples that illustrate key programming techniques. Thus this text is not intended to be a sourcebook of "useful" machine language programs. Instead it is designed to explain how machine language commands work so that you can understand more complex programs given in other texts and write your own machine language programs. In addition to giving you essential programming techniques, this text is designed to give you independence—the ability to experiment with your Apple and figure out for yourself whatever it is you need to know. Anyone who masters the material presented here should have no trouble going on to more advanced machine language programming techniques.

*Robert D. Rosen*

# Preface

The first of the two volumes of this work is a general introduction to the study of the history of the world, and the second is a more detailed account of the history of the world, from the beginning of the world to the present time. The first volume is intended for the general reader, and the second for the student of history. The first volume is divided into two parts, the first part dealing with the history of the world from the beginning of the world to the present time, and the second part dealing with the history of the world from the present time to the future. The second volume is divided into two parts, the first part dealing with the history of the world from the beginning of the world to the present time, and the second part dealing with the history of the world from the present time to the future. The first volume is intended for the general reader, and the second for the student of history.



# Contents

|                                                                                                       |           |
|-------------------------------------------------------------------------------------------------------|-----------|
| <b>Preface</b>                                                                                        | <b>v</b>  |
| <b>1 NUMBER SYSTEMS</b>                                                                               | <b>1</b>  |
| Review Questions                                                                                      | 4         |
| <b>2 INTRODUCTION TO MACHINE LANGUAGE<br/>PROGRAMMING: THE APPLE MONITOR AND<br/>ADDRESSING MODES</b> | <b>6</b>  |
| Types of Memory                                                                                       | 6         |
| Entering and Leaving the Monitor Program                                                              | 7         |
| Examining Memory                                                                                      | 7         |
| Changing the Contents of Memory                                                                       | 9         |
| The Accumulator                                                                                       | 10        |
| Addressing Modes                                                                                      | 13        |
| Assembly Listings                                                                                     | 16        |
| Editing Machine Language Programs Using the<br>Monitor                                                | 17        |
| Review Questions                                                                                      | 22        |
| Problems                                                                                              | 23        |
| <b>3 SOME USEFUL INTRINSIC SUBROUTINES</b>                                                            | <b>25</b> |
| X and Y Registers                                                                                     | 25        |
| Plotting Points                                                                                       | 26        |
| Zero-Page Memory                                                                                      | 27        |
| Addresses of Machine Language Subroutines                                                             | 28        |
| Other Graphics Subroutines                                                                            | 29        |
| Effect of Subroutines on the Accumulator,<br>X Register, and Y Register                               | 31        |
| ASCII Codes                                                                                           | 32        |
| Printing Characters in Machine Language                                                               | 33        |
| Printing Numerical Information                                                                        | 34        |
| Input from the Keyboard                                                                               | 35        |
| Random Numbers                                                                                        | 36        |
| The AND Command                                                                                       | 37        |
| Using the AND Command with the Get a Keystroke<br>Subroutine                                          | 39        |

|          |                                                                                                           |            |
|----------|-----------------------------------------------------------------------------------------------------------|------------|
|          | NORMAL, FLASH, and INVERSE in Machine Language                                                            | 41         |
|          | An Alternate Method of Displaying Text                                                                    | 45         |
|          | Absolute Indexed Addressing Mode                                                                          | 45         |
|          | Review Questions                                                                                          | 48         |
|          | Programming Problems                                                                                      | 49         |
| <b>4</b> | <b>MACHINE LANGUAGE LOOPS</b>                                                                             | <b>50</b>  |
|          | Doing HEX Arithmetic Using the Monitor. Negative Numbers                                                  | 50         |
|          | Saving Programs in Binary                                                                                 | 54         |
|          | Machine Language Loops                                                                                    | 56         |
|          | The Compare Command and the Processor Status Register                                                     | 56         |
|          | Branch Commands and the Relative Addressing Mode                                                          | 59         |
|          | Using Compare and Branch Statements Together                                                              | 65         |
|          | Unconditional Jumps                                                                                       | 73         |
|          | Subroutines in Machine Language                                                                           | 75         |
|          | A Step-by-Step Programming Example:<br>The Bubble Sort                                                    | 77         |
|          | Review Questions                                                                                          | 84         |
|          | Programming Problems                                                                                      | 86         |
| <b>5</b> | <b>NESTED LOOPS AND APPLE SOUNDS</b>                                                                      | <b>88</b>  |
|          | Using the Mini-assembler                                                                                  | 92         |
|          | Apple Sounds                                                                                              | 111        |
|          | Review Questions                                                                                          | 117        |
|          | Programming Problems                                                                                      | 119        |
| <b>6</b> | <b>ARITHMETIC</b>                                                                                         | <b>121</b> |
|          | Adding Two HEX Numbers                                                                                    | 121        |
|          | Adding HEX Numbers More than Two Digits Long                                                              | 124        |
|          | The Decimal Mode                                                                                          | 127        |
|          | Reading in Two-Digit Numbers from the Keyboard                                                            | 129        |
|          | Subtraction                                                                                               | 138        |
|          | Multiplication                                                                                            | 140        |
|          | Use of ASL and ROL                                                                                        | 146        |
|          | ROR Command                                                                                               | 148        |
|          | Review Questions                                                                                          | 150        |
|          | Programming Problems                                                                                      | 152        |
| <b>7</b> | <b>SHAPE TABLES</b>                                                                                       | <b>153</b> |
|          | Plotting Vectors                                                                                          | 153        |
|          | Binary Codes for Plotting Arrows and Placement of the<br>Binary Codes into the Byte of a Shape Definition | 155        |
|          | Calculation of the HEX Numbers Making Up the Shape<br>Definition                                          | 158        |



## **Contents**

**ix**

|                                                                                          |            |
|------------------------------------------------------------------------------------------|------------|
| The Index                                                                                | 161        |
| Entering the Shape Table into Apple's Memory                                             | 162        |
| Using Shape Tables—DRAW, XDRAW, ROT, and SCALE                                           | 163        |
| Step-by-Step Example: Constructing and Using a Shape Table                               | 167        |
| Motion Graphics—Program 1                                                                | 170        |
| Computer Art—Program 2                                                                   | 171        |
| Review Questions                                                                         | 171        |
| Programming Problems                                                                     | 173        |
| <br>                                                                                     |            |
| <b>8 INDIRECT ADDRESSING AND SOME MISCELLANEOUS TOPICS</b>                               | <b>175</b> |
| Indirect Addressing                                                                      | 175        |
| Preindexed Indirect Addressing Mode                                                      | 178        |
| Postindexed Indirect Addressing Mode                                                     | 181        |
| Representation of Two-Dimensional Matrices                                               | 183        |
| Integration of Machine Language Programs with BASIC Programs—Use of POKE, PEEK, and CALL | 190        |
| Using POKE and PEEK to Work with Data Values Too Large to Fit into One Memory Location   | 193        |
| Debugging Programs                                                                       | 197        |
| Review Questions                                                                         | 199        |
| Programming Problems                                                                     | 201        |
| <br>                                                                                     |            |
| <b>APPENDICES</b>                                                                        |            |
| <br>                                                                                     |            |
| <b>A ANSWERS TO SELECTED REVIEW QUESTIONS</b>                                            | <b>203</b> |
| <b>B OP CODES</b>                                                                        | <b>206</b> |
| <b>C LOW-AND HIGH-RESOLUTION COLORS</b>                                                  | <b>218</b> |
| <b>D SUMMARY OF USEFUL BUILT-IN SUBROUTINES AND CALL LOCATIONS</b>                       | <b>220</b> |
| <b>E ASCII CODES</b>                                                                     | <b>224</b> |
| <b>F TEXT SCREEN MAP</b>                                                                 | <b>228</b> |
| <br>                                                                                     |            |
| Glossary                                                                                 | 230        |
| Index                                                                                    | 237        |





# ***Apple Machine Language***

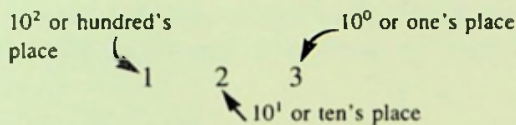
Apple Machine Language



# Chapter 1

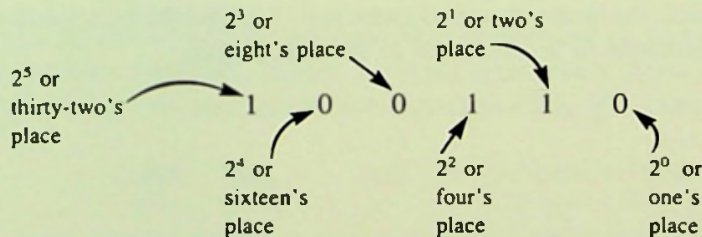
## Number Systems

Before we can start to discuss machine language programs themselves, the reader must have a working knowledge of number systems. The decimal or base 10 system is used for everyday computations. It consists of ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The value of any digit depends on the place in which it is found. For example, in 123 we have



"1"'s value is  $1 \times 100$  because the "1" is in the hundred's place, but the value of "3" is  $3 \times 1$  or 3 because the "3" is in the one's place. The value of each place increases by a factor of 10 as we move to the left.

It is possible to consider bases other than 10. Two of these are important to the machine language programmer: base 2 (binary) and base 16 (hexadecimal or HEX). In the binary system, there are two digits: 0 and 1. The value of each place increases by a factor of 2 as we move to the left. For example, consider the binary number 100110.



To convert this number to its decimal equivalent, we multiply each digit in the number by its corresponding place value and add all the results:

$$\begin{aligned}
 100110_2 &= 1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 \\
 &= 38_{10}
 \end{aligned}$$

Note the use of subscripts on 100110 and 38 to indicate the base of the numerals. This process for converting from binary to decimal is known as *expanding the binary number*. You can see that even small numbers in decimal have long binary representations. Thus binary is very awkward to work with. The binary number system is important, however, because it is the language that computers understand. Conceptually a computer can be thought of as a large number of switches, each of which is either on or off. Because each switch has only two possible states, the binary number system is ideal for representing the switches.

Apple's memory is organized into a large number of *locations* that are analogous to pigeonholes in which we can store information. In BASIC, we gave a name to each location we used. For example,

10 X = 6

puts a 6 in a location we refer to as X in the rest of our program. In order to store the digits 0 through 9 or characters (in the case of *string variables*), each memory location must consist of more than just one *switch*. In fact, eight switches are grouped together, so that each memory location can store the equivalent of an eight-digit binary number. This means that any number from

00000000<sub>2</sub>

0 in decimal

to

11111111<sub>2</sub>

255 in decimal

can be stored in a single location. Thus we have 256 possible values that can be stored in any location. In machine language, the various commands, letters of the alphabet, and other keyboard characters are each given a numerical code. With the 256 possible values for any memory location, we can store any character we wish or program the computer with dozens of different machine language commands.

As previously mentioned, the binary system is awkward. Consider the binary codes for two machine language commands to be discussed in the next chapter:

10101001

10101101

Even a short program might have 20 or 30 such commands. It is difficult for humans to work with such codes because they are all so similar. It would also take a long time to type in these codes, because each one is so long. For these reasons, the hexadecimal (HEX) system is used. The HEX system is best thought of as a shorthand way of representing long binary numbers. In hexadecimal (base 16), there are 16 different digits. The following table shows the representations for these 16 digits in HEX, decimal, and binary.



| HEX | Decimal | Binary |
|-----|---------|--------|
| 0   | 0       | 0000   |
| 1   | 1       | 0001   |
| 2   | 2       | 0010   |
| 3   | 3       | 0011   |
| 4   | 4       | 0100   |
| 5   | 5       | 0101   |
| 6   | 6       | 0110   |
| 7   | 7       | 0111   |
| 8   | 8       | 1000   |
| 9   | 9       | 1001   |
| A   | 10      | 1010   |
| B   | 11      | 1011   |
| C   | 12      | 1100   |
| D   | 13      | 1101   |
| E   | 14      | 1110   |
| F   | 15      | 1111   |

Notice that the letters A through F are used in HEX to represent the decimal numbers 10 to 15. Extra characters are needed because HEX has 16 different digits instead of the 10 we are familiar with in decimal. You would not want to use

$\frac{\quad}{\quad} \frac{10}{\quad}$   
 sixteen's place      one's place

to represent 10 in HEX because each place can hold only one digit. It would be too easy to confuse

$\frac{\quad}{\quad} \frac{10}{\quad}$        $\frac{1}{\quad} \frac{0}{\quad}$   
 sixteen's place      one's place      with      sixteen's place      one's place

so it was decided to use A to represent 10, B to represent 11, and so on. It is easy to change a binary number into HEX. Simply group the eight binary digits into blocks of four and use the table to convert each block from binary to HEX. For example,

$\frac{1010}{\quad} \quad \frac{1001}{\quad}$       and       $\frac{1010}{\quad} \quad \frac{1101}{\quad}$   
 A      9      A      D

so

$$10101001_2 = A9_{\text{HEX}} \quad \text{and} \quad 10101101_2 = AD_{\text{HEX}}$$

The table can also be used to convert from HEX back into binary. Simply replace each HEX digit by its binary representation. For example, to convert  $8D_{\text{HEX}}$  into binary, note that

$$8_{\text{HEX}} = 1000_2 \quad \text{and} \quad D_{\text{HEX}} = 1101_2$$

so

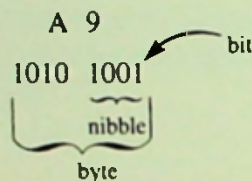
$$8D_{\text{HEX}} = 10001101_2$$

We will use HEX to enter all our machine language commands. Instead of working with eight binary digits, we will have to enter only two HEX digits into each memory location.

Before leaving the topic of number systems, some terminology concerning binary and HEX should be discussed. When programmers refer to a *bit*, they mean a single binary digit. Thus each memory location in the Apple holds 8 bits. Eight bits are referred to as a *byte*. Thus each memory location holds 1 byte of information and two HEX digits are needed to represent 1 byte. One HEX digit is referred to as a *nibble*, so a nibble is half a byte. Each of Apple's memory locations will contain 2 nibbles. In summary,

$$8 \text{ bits} = 2 \text{ nibbles} = 1 \text{ byte}$$

Eight binary digits = two HEX digits = contents of one memory location



In the next chapter, we discuss the use of Apple's monitor. The monitor program is used to store HEX codes in memory and to examine the contents of memory. In the monitor, all numbers are represented in HEX. To avoid mistakes, it is important to become comfortable with the HEX number system. For example, the memory location listed after 319 will be 31A, not 320. The location located eight places beyond 307 is 31F, not 315. After a little practice, you will learn to "think in HEX," a skill quite valuable for the machine language programmer.

### Review Questions

- Convert each binary number to HEX.  
(a) 10110010      (b) 00011110      (c) 00111010
- Convert each HEX number to binary.  
(a) 3D      (b) 8C      (c) AD
- Convert each HEX number to decimal.  
(a) 17      (b) 2A      (c) A9  
(Hint: The left digit is in the sixteen's place and the right digit is in the one's place.)



4. Study the following procedure.
  - (a) Divide 425 by 16 to obtain quotient 26, remainder 9.
  - (b) Divide the quotient 26 by 16 to obtain a new quotient 1 and a new remainder 10.
  - (c) Divide the last quotient by 16 to obtain the quotient 0, remainder 1.
  - (d) Write the remainders down next to each other, using HEX notation for each remainder. Write the last remainder down first: 1A9.  
Observe that

$$1A9_{\text{HEX}} = 1 \times 256 + 10 \times 16 + 9 = 425_{10}$$

Thus, you can convert a decimal number into HEX by successive divisions by 16 (stop when the quotient in any step becomes zero). Using this method, convert each of these decimal numbers to HEX.

- (a) 169              (b) 928

## ***Chapter 2***

# ***Introduction to Machine Language Programming: The Apple Monitor and Addressing Modes***

The monitor program can be used to examine the contents of any of Apple's memory locations. It allows the user to change the contents of memory and to move blocks of data in memory. These last two capabilities will enable us to enter and edit machine language programs. In this chapter, we first learn some monitor commands. These commands will then be used to enter and edit a simple machine language program.

### **Types of Memory**

Apple's memory is divided into two categories: read only memory (ROM) and random access memory (RAM). ROM stores information vital to the computer's functioning—Applesoft, the monitor program, and various subroutines. Information in ROM does not go



away when the the Apple is turned off. We can examine ROM using the monitor, but we cannot (as the name "read only" implies) change the contents of this memory. Thus the contents of ROM are built into the Apple.

On the other hand, random access memory (RAM) is available for our use. We can read this memory and change its contents. The amount of RAM depends on the particular Apple. When we say we have a "16K machine," we mean we have (approximately) 16,000 bytes of RAM. Similarly a "48K machine" refers to a machine with 48,000 bytes of RAM.

We will be entering our machine language programs in locations \$300 to \$3FF. The currency symbol \$ indicates that a number is being given in HEX, so \$300 means  $300_{16}$ , and 300 without the "\$" would refer to location  $300_{10}$ . Although I will leave off the "\$" in some instances, all location addresses will be understood to be in HEX unless otherwise stated.

Locations other than \$300-\$3FF are suitable for storing machine language programs. In fact, I have stored programs in locations \$800 through \$BFFF on my 48K Apple. However, some of these locations are used for other purposes (graphics, DOS, etc.) and your program could get "wiped out" if Apple tried to use these locations at the same time you are using them. In any case, you should never use locations below \$300, between \$400 and \$7FF, or above \$C000 to store programs, as these locations are used for other purposes. Locations \$300 to \$3FF do not interfere with any of Apple's other functions, which is why they are ideal for storing short machine language programs.

## Entering and Leaving the Monitor Program

If you are in Applesoft, you can enter the monitor by typing CALL-151 (RETURN). An asterisk will appear along with a flashing cursor. The asterisk is the monitor's prompt symbol, just as the square bracket is Applesoft's prompt character.

To leave the monitor, type Control-C (RETURN). Applesoft's prompt will reappear.

## Examining Memory

Before we discuss the monitor commands for examining memory, enter the monitor program on your Apple and type the following. Apple will display the asterisk prompt symbol. You should type everything else exactly as shown.

```
*300:01 02 03 04 05 06 07 08 09 0A (RETURN)
*:0B 0C 0D 0E 0F 10 11 12 13 14 15 (RETURN)
*:16 17 18 19 1A 1B 1C 1D 1E 1F 20 (RETURN)
*
```

Now there will be something in memory for us to examine. (In case you are wondering, we just filled location 300 with 01, location 301 with 02, location 302 with 03, . . . and location 31F with 20. More will be said about changing memory's contents later.)

To examine a single location of memory, just type the address (location) in which you are interested and hit the RETURN key. For example,

```
*300 (RETURN)
```

yields

```
0300-01
```

```
*
```

which means the byte 01 is stored in location 0300. See if you can guess what the contents of \$309 are. Verify your answer by typing

```
*309
```

You should spend a few minutes experimenting with the memory inspection feature. You will become more comfortable with the monitor and the HEX number system. Answer questions for yourself, such as, "What address contains 0F?". Guess an address, and then examine its contents to see if it really does contain 0F. Make sure you understand why 0F is stored in \$30E before going on.

To examine many memory locations, we do not have to type each address individually. The monitor command

```
* 300.31F
```

will examine a *range* of memory from location 300 to 31F (inclusive). In general,

```
* Addr 1 . Addr 2 (RETURN)
```

can be used to examine all memory locations between Addr 1 and Addr 2 inclusive (Addr stands for a HEX address).

You should now type the command.

```
* 300.31F
```

Apple will respond:

```
This display 0300- 01 02 03 04 05 06 07 08
of memory is 0308- 09 0A 0B 0C 0D 0E 0F 10
known as a  0310- 11 12 13 14 15 16 17 18
"memory    0318- 19 1A 1B 1C 1D 1E 1F 20
dump."
```



It is important to note that Apple does not label each byte with an address as it did when we examined a single memory location. To save space, Apple displays 8 bytes per row and gives the address of only the first byte in each row. A space separates each byte in the row. In this format, the easiest way to find the contents of an address is to start at the beginning of the appropriate row and mentally count in HEX until the desired address is reached. For example, in

|       |     |     |     |     |     |          |
|-------|-----|-----|-----|-----|-----|----------|
|       |     | 309 | 30B | 30D | 30F |          |
|       |     |     |     |     |     |          |
| 0308- | 09  | 0A  | 0B  | 0C  | 0D  | 0E 0F 10 |
|       |     |     |     |     |     |          |
|       | 308 | 30A | 30C | 30E |     |          |

location \$30E contains 0F as shown above. Next try typing the command

\* 305.317

Apple responds:

```
0305- 06 07 08
0308- 09 0A 0B 0C 0D 0E 0F 10
0310- 11 12 13 14 15 16 17 18
```

This example illustrates the following. With the exception of the first row, all rows will begin in an address ending with 8 or 0. The first row contains the tail end of the 300 row, which is why it contains only 3 bytes. Monitor will always display 8 bytes per row, unless you do not ask for the complete first (or last) row. The orderly format of the monitor makes it easier to find the contents of an address that is not the first member of its row and therefore will not have an address label explicitly displayed.

Another method is to hit the RETURN key after examining memory either individually or by use of the memory range method. Each time the RETURN key is hit, a row of bytes is displayed. Each row starts where the last one left off. For example,

|          |            |                               |                        |
|----------|------------|-------------------------------|------------------------|
| * 303    | gives      | 0303- 04                      | (finish the "tail end" |
| * RETURN | then gives | 05 06 07 08                   | of the 300 row)        |
| * RETURN | then gives | 0308- 09 0A 0B 0C 0D 0E 0F 10 |                        |
| * RETURN | gives      | 0310- 11 12 13 14 15 16 17 18 |                        |

and each additional RETURN will give a complete line of 8 bytes.

## Changing the Contents of Memory

The colon is used after an address is typed to tell the monitor to change the contents of the address. For example,

300:22

means "store \$22 in location \$300." You should actually try this command and then examine location \$300 to verify that \$22 was stored in \$300.

We can change the contents of more than one location at a time. Type the address of the first location you wish to change and follow it by a colon. Next type the contents you wish to go into the address just typed. Additional bytes may then be typed with spaces between them. These bytes will be placed in consecutive memory locations following the first location. For example,

08 is placed  
in \$321  
 \* 320:05 08 19 2B      2B is placed in \$323  
 05 is placed      19 is placed  
in \$320            in \$322

This was the method used at the beginning of the chapter to fill memory locations 300 to 31F. In practice, it is a good idea to load memory using the same format that monitor uses to display the contents of memory. Begin your rows with addresses ending in 0 or 8 (when possible). For example, to fill locations 303 to 30F, use

```
* 303:01 02 03 04 05
* 308:06 07 08 09 0A 0B 0C 0D
```

instead of

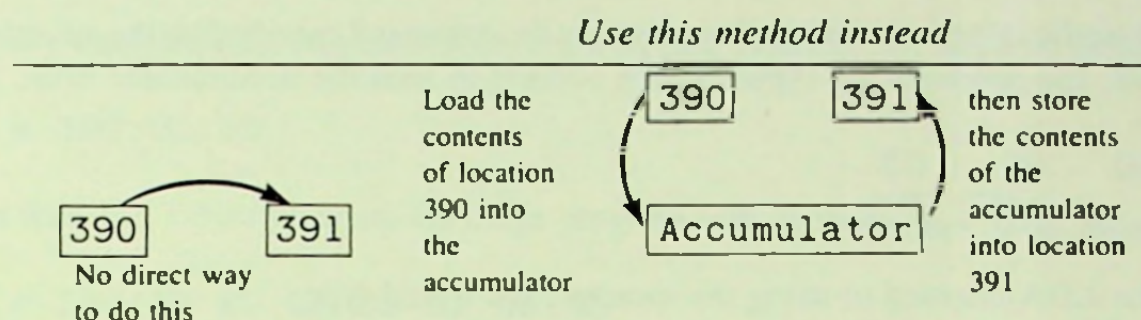
```
* 303:01 02 03 04 05 06 07 08 09 0A 0B 0C 0D
```

Typing in your data in the 8-bytes-per-row format makes it easier to detect mistakes. It is also less confusing than trying to type long strings of bytes.

### The Accumulator

We are now at the stage where we can actually enter bytes representing a machine language program. Perhaps the most important memory location in machine language is the *accumulator*. Although most memory locations are referred to by number (location \$300 or location \$3F2, and so on), the accumulator is not. In almost all respects, the accumulator behaves just as any other storage location, however. It is used mainly to move information from one location to another. Arithmetic is also done in the accumulator. Here we examine the accumulator's role in moving data from one memory location to another. Suppose we need to move the contents of location 390 into location 391. In machine language, there is no direct way to do this. Instead we use the accumulator as a *go-between* as illustrated in the following.





We first examine the commands "Load the Accumulator" (LDA) and "Store the Accumulator" (STA). The Load the Accumulator command is used to put a copy of what is in a memory location into the accumulator. The data remain in the memory location as well, so the command LDA is a "copy" command more than a load command. The command Store the Accumulator copies the contents of the accumulator into a memory location. When we use STA, the data we started with remain in the accumulator, so STA is also a copy command.

To use the LDA command, you have to know the HEX code for LDA, which can be found in the table in Appendix B. It is best not to memorize these HEX codes. When you write machine language programs, you will first write your programs on paper using mnemonic codes such as LDA and STA. Then you will look up the HEX codes as needed, and type these into Apple's memory using the monitor program (the process of converting from the mnemonic codes of assembly language to the HEX codes of machine language is called *assembling the program*). When the mini-assembler is introduced, you will be able to enter a program in assembly form—that is, by typing in mnemonic codes such as LDA. For now, however, we will enter the HEX codes for machine language commands using the monitor, after looking up the codes in a table.

The HEX code for LDA is AD. When using the monitor, it is not enough just to type AD, because the computer will not know where you want to load the accumulator from. You will use the complete machine language code shown below.

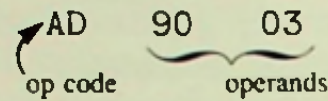
| <i>Machine Language Form</i> | <i>Mnemonic or Assembly Form</i>                         |
|------------------------------|----------------------------------------------------------|
| AD 90 03                     | LDA 390                                                  |
|                              | Load the      from      location<br>accumulator      390 |

Note that the address 0390 is entered into the monitor "backward" as 90 03 instead of 03 90. Also note that an address is four digits long and so consists of 2 bytes. The *most significant byte* (MSB) of an address is the two digits of highest place value (03 here). The *least significant byte* (LSB) consists of the two digits in the one's and sixteen's place in the address (90 in this example). When entering an address as part of a program, the MSB of the address is always put in the higher memory location.

To translate LDA 390 into machine language, we must use 3 bytes. The first byte is called the operation code or *op code* (which is AD in our example), and is used to select the instruction you wish to use. The other 2 bytes are called *operands*. Operands give the



computer specific information needed to carry out the command specified by the op code. In our example, the operands tell Apple which address to load the accumulator from.

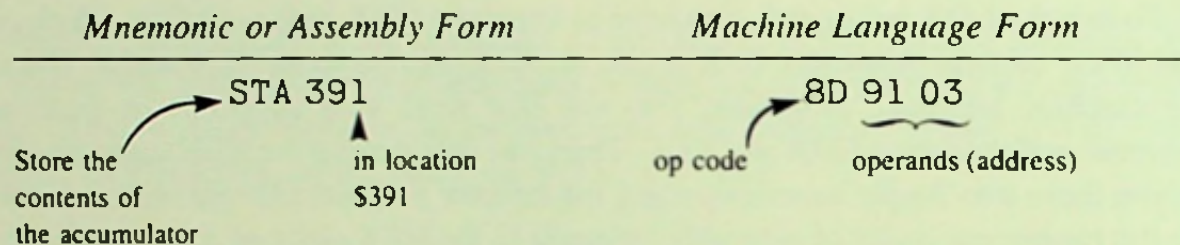


To enter the LDA instruction using the monitor, we would type

```
* 300:AD 90 03
```

and the instruction would be stored beginning in location \$300.

The command Store the Accumulator is also a 3-byte command, consisting of an op code 8D and a 2-byte operand that tells Apple where you want the contents of the accumulator stored.



Before we can run our first machine language program, one more command needs to be mentioned—RTS, the mnemonic abbreviation for "ReTurn from Subroutine." The op code for this command is 60 and the command requires no operands. RTS returns control to the monitor program after your machine language program has been executed. RTS is required as the last statement in all machine language programs. Our first complete machine language program will consist of the commands introduced so far: LDA, STA, and RTS, and is shown in the following.

*Machine Language Form*  

```
* 300:AD 90 03 8D 91 03 60
```

| Assembly Form | Comments                                                                                                                                       |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| LDA 390       | { Loads (copies) the contents of 390 into the accumulator. A copy of the original data will exist in location 390 and also in the accumulator. |
| STA 391       | { Store (copy) the contents of the accumulator in location 391. A copy of the original data will now exist in 390, 391, and the accumulator.   |
| RTS           | Return control to the monitor program.                                                                                                         |



Try the following on your Apple. Put a 01 into location 390 and a 00 into 391 using

```
* 390:01 00
```

Then load the 7-byte machine language program into memory by typing

```
* 300:AD 90 03 8D 91 03 60
```

If you type

```
* 300G
```

the program will be executed. "G" stands for GO. Apple will begin executing your program starting in location \$300 and end when the RTS instruction is encountered. The monitor prompt (an asterisk) will appear as soon as the program is finished running, which will be just about instantly. How can you be sure that the program worked? Type the monitor command

```
* 390.391
```

Apple will respond with

```
0390-01 01
```

showing that the datum originally in location 390 was copied into location 391.

### **Addressing Modes**

Before describing the editing of machine language programs, let us write a machine language program equivalent to

```
10 M=1  
20 N=2  
30 SAFE=M  
40 M=N  
50 N=SAFE  
60 END
```

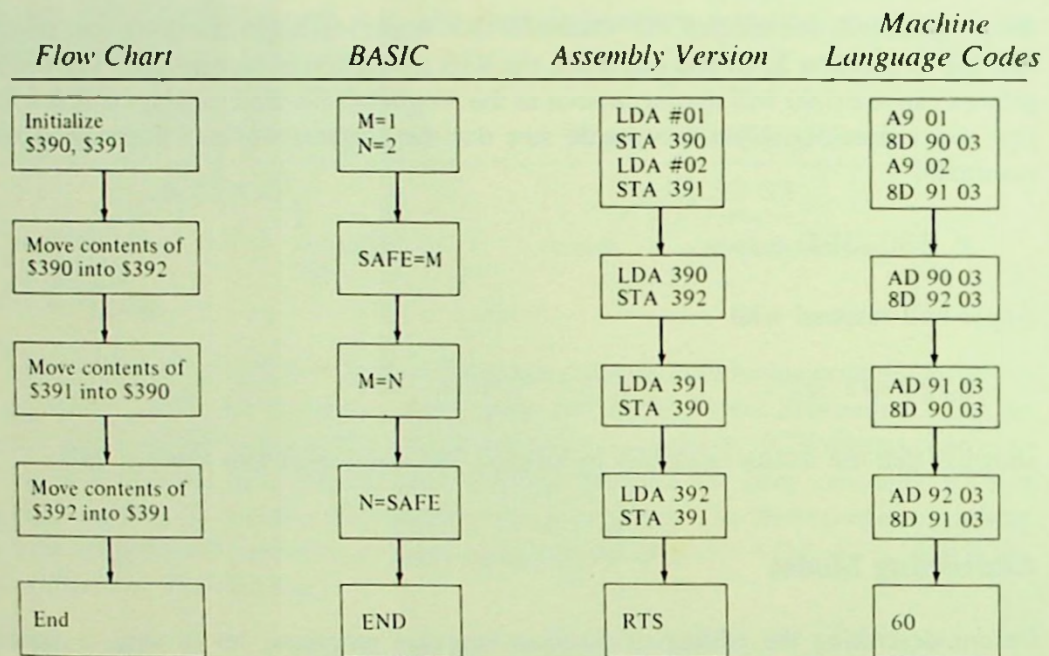
In other words, we will initialize the values of two memory locations and then switch the contents of the two locations. In writing any machine language program, the following steps are helpful.

1. Write a flowchart or block diagram outlining the major steps to be performed.

2. Write the assembly (mnemonic) form of the program by translating your flow-chart directly into assembly language instructions.
3. Look up the HEX op codes corresponding to the assembly version and enter the op codes and the corresponding operands using the monitor. (Op codes can be found in Appendix B.)

In the following.

location \$390 corresponds to M  
 location \$391 corresponds to N  
 location \$392 corresponds to SAFE



The program would be entered using the monitor. Enter the HEX codes 8 bytes per line.

```
* 300:A9 01 8D 90 03 A9 02 8D
* 308:91 03 AD 90 03 8D 92 03
* 310:AD 91 03 8D 90 03 AD 92
* 318:03 8D 91 03 60
```

Type

```
* 300G
```

to run the program. To see if this program does what you expect it to do, use the monitor command



\* 390 .392

to examine the contents of memory locations 390, 391, and 392.

All blocks in the flowchart, except the first, contain machine language codes previously explained. In the first block, the LDA command is being used in a new *addressing mode*. Consider the command LDA 390 discussed in the previous section. Apple had to go to location 390, look up the contents stored there, and copy the contents into the accumulator to execute this command. Because the 390 refers to an *address*, we say LDA is being used in the *absolute addressing mode*. In other words, the accumulator is not being loaded with 390; it is being loaded with the contents of address 390. All absolute addressing mode commands require Apple to refer to data stored in the address specified by the command's operands. All absolute addressing mode commands are 3 bytes long—1 byte for the op code and 2 bytes for the operand (address).

Notice that to initialize 390 in the sample program of this section, we first put 01 into the accumulator and then stored the accumulator's contents in location 390 (using the two commands LDA #01 and STA 390). When we loaded the accumulator with 01, it was not necessary for Apple to look up the 01 in a memory location. The operand 01 represented actual data to be loaded into the accumulator, not an address where the data could be found. When LDA is used in this way, we are using the *immediate addressing mode*. (Do not confuse the immediate addressing mode in machine language with the immediate execution mode of Applesoft—the two modes have nothing to do with one another.) In LDA #01, the operand 01 is immediately placed into the accumulator—no addresses are referred to at all. To signify that 01 represents data rather than an address, the # symbol is used in the assembler (mnemonic) abbreviation. Make sure you understand the difference between

|                                 |     |                                                      |
|---------------------------------|-----|------------------------------------------------------|
| LDA #01                         | and | LDA \$01                                             |
| Load the accumulator<br>with 01 |     | Load the accumulator<br>from memory location<br>\$01 |

All immediate addressing mode commands are 2 bytes long—1 byte for the op code and 1 byte for the operand (data).

From the preceding example, you should see that LDA is a mnemonic for a machine language instruction, but this mnemonic will have a different op code for each of its possible addressing modes. When you look up the HEX op code for LDA, make sure you choose the op code corresponding to the addressing mode you require. You should also realize that commands other than LDA can be used in more than one addressing mode. The conceptual distinction between absolute and immediate addressing modes is the same, regardless of the particular command being considered. This will become clearer as you learn more commands.

There is one other addressing mode to which you have already been exposed—the *implied addressing mode*. An example of a command being used in the implied addressing mode is RTS. In machine language, this command consists of only 1 byte, the op code 60. Although the command LDA required additional information (an address or data), the command RTS is a self-contained unit. The op code itself gives Apple enough information

to carry out the command. Commands such as RTS, which require no operands, are implied mode commands. They are all 1 byte long.

### Assembly Listings

When we enter long machine language programs by typing HEX numbers using the monitor, it is easy to make mistakes. And it is difficult to look at a group of two-digit HEX numbers and debug a program that does not work. To help overcome these problems, the monitor command L can be used. For the program of the last section, if you type \* 300L (L stands for List), you get an assembly language listing of your program—one command per line (the process of converting the HEX codes of machine language into the mnemonic codes of assembly language is referred to as *disassembling the machine language program*). Up to 20 assembly language commands can be displayed at once. The listing of the example program would look like the following.

```
0300- A9 01      LDA #$01
0302- 8D 90 03   STA $0390
0305- A9 02      LDA #$02
0307- 8D 91 03   STA $0391
030A- AD 90 03   LDA $0390
030D- 8D 92 03   STA $0392
0310- AD 91 03   LDA $0391
0313- 8D 90 03   STA $0390
0316- AD 92 03   LDA $0392
0319- 8D 91 03   STA $0391
031C- 60         RTS
```

The FF's and ???'s that appear under the RTS command in the real monitor listing are "garbage" to be ignored; your program took only half a screen to list. Notice that each command appears on its own line. The monitor program groups related bytes automatically. For example, A9 01 are displayed on a single line since these 2 bytes make up the command LDA #\$01

The numbers in the first column are not line numbers. These numbers represent the memory location of the first byte (the op code) in each machine language command. These memory locations will be useful when we discuss editing.

It should be noted that you may begin a listing anywhere in your program. For example, the command

```
* 313L
```

would list your program beginning with the command whose op code is stored in \$313.

Unlike BASIC, if you list a long program, the listing will not *scroll* off the top of the screen. The monitor stops as soon as the screen is full of the first batch of machine language commands. If your program is too long to be listed on one screen, you do not have to type



the address of the command following the last one shown to get the next screenful of commands. Instead, type

\* L

without an address. The monitor will continue your listing with a new screen of instructions. The listing will automatically pick up where the last screen left off.

### **Editing Machine Language Programs Using the Monitor**

Suppose the command stored in location 300 was typed, by mistake, as A0 instead of A9 in our example program. If the program were listed using

\* 300L

the results would show up as

```
0300- A0 01      LDY #$01
0302- 8D 90 03    STA $0390
0305- A9 02      LDA #$02
      .
      .
      .
```

instead of as

```
0300- A9 01      LDA #$01
0302- 8D 90 03    STA $0390
0305- A9 02      LDA #$02
      .
      .
      .
```

The first command is LDY ("LoaD the Y register"—a command used in the next chapter) instead of LDA, because we typed the wrong op code. Editing this program is easy because the correct command takes up the same number of bytes as the incorrect command. We merely type

\* 300:A9

which changes the incorrect data stored in 300 to the desired value. As in BASIC, if we fill a memory location with new information (A9), it "pushes out" the old information (A0). Notice that the number 0300 given in the left-hand column of the monitor assembly listing gives the address of the first byte of the command listed on the line. Had we wished to

change the 01 instead of the A0, we would have altered location 0301. Also notice that if we want to replace a 2-byte command by a 3-byte command, we are out of luck. If we try to replace the LDY #01 command with, say, LDA 390 by typing

```
* 300:AD 90 03
```

we will alter locations 300, 301, and 302. But location 302 already contains an instruction we do not want to change. In BASIC, changing commands causes no problems. You can replace any line by typing a new one with the same line number as the old one. It does not matter whether the new line is longer than the old one. In machine language, however, the numbers listed to the left are not line numbers. They show in which memory location a command is stored. You can replace the contents of memory easily enough, but you cannot insert a 3-byte command where a 2-byte command used to be located. In BASIC, we could add an instruction between two already existing instructions by giving the new instruction a line number between the two already existing line numbers. Notice that in machine language there is no direct way to insert a command between two already existing commands. There are no empty memory spaces between existing machine language commands in which to store the new instruction.

We next examine a method that can be used to make room for new commands between two already existing commands in a machine language program. The same method can be used to make room so that 2-byte commands can be replaced by 3-byte commands.

The key to making room for new commands is the monitor Move command. The command's general form is

```
addr 1 < addr 2.addr 3 M
```

This command takes the block of data stored in locations addr 2 to addr 3 (inclusive) and copies this block in the location beginning with address addr 1. For example, the command

```
* 800 < 305.325M
```

copies all the information stored in locations 305 to 325 inclusive in the block of memory beginning with location 800. This means that locations 800 to 820 will have the same contents as locations 305 to 325 after the Move command is executed.

Returning to our example program, suppose we need to insert the command STA 395 between the commands STA 390 and LDA #02:

|       |          |            |                 |
|-------|----------|------------|-----------------|
| 0300- | A9 01    | LDA #\$01  |                 |
| 0302- | 8D 90 03 | STA \$0390 |                 |
| 0305- | A9 02    | LDA #\$02  | ← Insert        |
| 0307- | 8D 91 03 | STA \$0391 | STA 395<br>here |

This can be accomplished through the following steps.

First, copy the section of the program that begins with the instruction that is to follow the command you want to insert.



\*800 < 305.325M

This command copies the entire program from LDA#02 to the end. We start copying with the instruction stored in 305, since we want LDA#02 to follow the new STA 395 command in the corrected version of the program.

Next, insert the new command:

\*305: 8D 95 03

You will be destroying the LDA#02 command, but a copy of it is safely stored in locations 800 to 820.

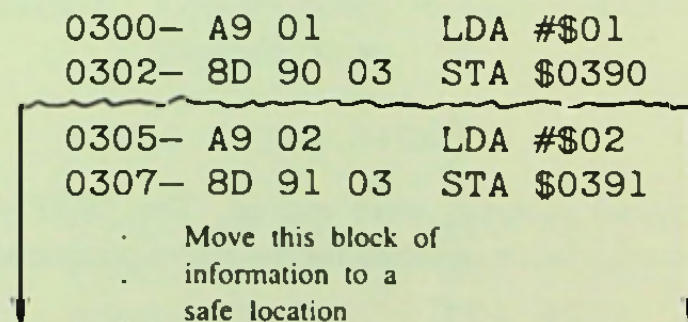
Finally, "splice" the part of the program you moved in the first step back onto your program. Be careful to add the rest of the program to the block of memory immediately following the STA 395 command you just added.

\*308 < 800.820M

The section of the program moved in the first step is added on to the first part of the program that was never moved. Notice that address 308, not address 305, is used as the destination of the move command. Although the section of the program that was moved used to be stored beginning in location 305, the new STA 395 command now occupies that location. The first available location after the STA 395 command is location 308.

Schematically the three steps used to edit our program were:

STEP I. \* 800 < 305.325M



STEP II. \* 305:8D 95 03

```

0300- A9 01      LDA #$01
0302- 8D 90 03   STA $0390
0305- 8D 95 03   STA $0395
  
```

Insert the new command

STEP III. \* 308 < 800.820M

|       |          |            |
|-------|----------|------------|
| 0300- | A9 01    | LDA #\$01  |
| 0302- | 8D 90 03 | STA \$0390 |
| 0305- | 8D 95 03 | STA \$0395 |
| <hr/> |          |            |
| 0308- | A9 02    | LDA #\$02  |
| 030A- | 8D 91 03 | STA \$0391 |

Splice the block originally  
moved back onto your  
program

In step III, notice that the LDA #\$02 command used to be stored in locations 305 and 306. It is now stored in locations 308 and 309. The rest of the program is also "pushed up" three storage locations.

There are two additional points that need to be made here. First, some may wonder why step I moved the last part of the program to the 800 location. Why not use

\* 308 < 305.325M

to "move the program up three spaces"? The monitor Move command can behave very unpredictably if the address mentioned to the left of the arrow is in the memory range mentioned on the right (and 308 is between 305 and 325). It is best to move the last part of your program to a location far away from the first part, which is not being moved, to avoid problems. The locations 400 to 7FF are not available for our use, so location 800 was chosen.

Some may also wonder why the command

\*800 < 305.31CM

was not used in step I. By using

\*800 < 305.325M

data beyond the RTS command of the original program were moved. The \*800 < 305.325M command was used to simplify the calculation required for the move command of step III. Suppose the command

\*800 < 305.31CM

had been used. The Move command would have copied the memory range 305 to 31C into locations 800 to 817 (since  $31C - 305 = 17$  and  $800 + 17 = 817$ ). Then the Move command in step III could have been

\*308 < 800.817M



Notice that in the Move command  $800 < 305.325M$ , the difference  $\$325 - \$305 = \$20$  is easy to find. This makes it easy to compute the proper memory range for the Move command in step III. Since the difference between the starting and ending addresses in the memory range of step I was  $\$20$ , we can say the difference between the starting and ending addresses in the memory range to be moved in step III must also be  $\$20$ . Therefore the memory range to be moved in step III must be  $\$800$  to  $\$800 + \$20$ , or  $800.820$ . To summarize, by moving a few more bytes than we had to in step I, we can simplify the work of step III.

We can also use the monitor Move command to replace long commands by short ones. For example, again consider the original listing of our example program.

```
0300- A9 01      LDA #$01
0302- 8D 90 03   STA $0390
0305- A9 02      LDA #$02
0307- 8D 91 03   STA $0391
030A- AD 90 03   LDA $0390
030D- 8D 92 03   STA $0392
```

·  
·  
·

Suppose we need to replace the LDA \$390 instruction by LDA #05. If we use the command

```
*30A:A9 05
```

we will have trouble, since the 03 in location 30C, which was originally part of the LDA 390 command, will not be "covered up" by the new command. This mixture of old and new commands in locations 30A to 30C will cause an execution error when the program is run. To replace a long command by a short one, follow essentially the same steps as before. The procedure is illustrated schematically in the following.

STEP I. Copy all those commands that appear after the one you wish to replace into a safe location. Type

```
* 800 < 30D.31DM
```

```
0300- A9 01      LDA #$01
0302- 8D 90 03   STA $0390
0305- A9 02      LDA #$02
0307- 8D 91 03   STA $0391
030A- AD 90 03   LDA $0390
```

```
030D- 8D 92 03   STA $0392
0310- AD 91 03   LDA $0391
```



·  
·  
·

Copy into locations  
800 to 810



STEP II. Replace the command LDA \$390 by the desired command. Type

\*30A: A9 05

```
0300- A9 01      LDA #$01
0302- 8D 90 03   STA $0390
0305- A9 02      LDA #$02
0307- 8D 91 03   STA $0391
030A- A9 05      LDA #$05
```

STEP III. Splice the program segment moved in step I back onto the part of the program not moved. Type

\*30C < 800.810M

```
0300- A9 01      LDA #$01
0302- 8D 90 03   STA $0390
0305- A9 02      LDA #$02
0307- 8D 91 03   STA $0391
030A- A9 05      LDA #$05
```

```
030C- 8D 92 03   STA $0392
030F- AD 91 03   LDA $0391
```

. Splice this part of the  
. program back onto the  
. rest of the program

Note that location 30C is used in the last Move command. This is so because location 30C is the first location that follows the command added in step II. Also note that 800.810 was chosen as the memory range for the last Move command. In step I we had a difference of  $\$31D - \$30D = \$10$  between the beginning and ending addresses of the range. Therefore in step III we choose an ending address of  $800 + \$10$  or 810, giving the required memory range 800.810.

You should practice adding, deleting, and changing commands in the sample program using the monitor. Machine language programs do not always run as originally written. The ability to edit programs using the monitor commands illustrated here is what makes the debugging process possible.

### Review Questions

1. Most short machine language programs will be stored in locations \$\_\_\_\_\_ to \$\_\_\_\_\_.
2. To enter the monitor program from Applesoft, type \_\_\_\_\_.



3. In the machine language command

AD 40 03

the first byte is called the \_\_\_\_\_ and the other 2 bytes are called \_\_\_\_\_.

4. All absolute addressing mode commands are \_\_\_\_\_ bytes long.  
5. The command RTS is an \_\_\_\_\_ addressing mode command.  
6. A machine language program is stored in locations \$300 to \$318. What monitor command could be used to obtain a listing of the program?  
7. What monitor command could you use to copy data stored in locations \$300 to \$340 into locations \$1000 to \$1040?  
8. Which command can be used to leave the monitor and enter Applesoft?

#### TRUE or FALSE

9. Programs that you write are stored in RAM.  
10. The monitor command

370- A9

is used to store \$A9 into memory location \$370.

11. All arithmetic operations that take place in machine language programs occur in the accumulator.  
12. The command

LDA #07

tells Apple to load the accumulator with the contents of memory location \$07.

13. Implied addressing mode commands have no operands.  
14. In a listing, the numbers on the left are line numbers.  
15. A machine language program is stored in locations \$300 to \$330. The monitor commands

314 < 313.330M  
313:DA

could be used to insert the byte DA in the middle of the program.

#### Problems

The following monitor commands are executed. These commands store into memory machine language instructions to which you will be introduced later.

```
*300:20 58 FC A0 1A A2 C1 8A
*308:29 7F 20 ED FD E8 88 D0
*310:F6 60 A9 8D 20 ED FD A2
*318:1A A0 C1 98 29 3F 20 ED
*320:FD C8 CA D0 F6 60
```

1. What bytes will monitor display if the command

\*307.31A

is given?

2. What byte is stored in each of these locations?  
(a) 305      (b) 30F      (c) 324
3. In what location is each of these bytes stored?  
(a) 8A      (b) 98      (c) 3F
4. Without retyping the entire program, delete the RTS command stored in location \$311, leaving the rest of the program intact.

A machine language program is entered using these monitor commands.

```
*300:AC 51 03 AD 50 03 8D 52
*308:03 2E 52 03 88 D0 AD 52
*310:03 20 DA FD 60
```

5. In this program, the byte FA needs to be inserted between the bytes D0 and AD. Show how this can be done without retyping the whole program.
6. The LDA 350 command stored in locations 303 to 305 is to be replaced by the command LDA #03. Show how this can be done without retyping the entire program.



# ***Chapter 3***

## ***Some Useful Intrinsic Subroutines***

In this chapter, we examine some of Apple's built-in features. In BASIC, we could run subroutines in a program by GOSUBing to the appropriate line number in our program. In Apple's ROM, there are certain useful functions that are analogous to subroutines. We can use these subroutines in our programs by jumping to the memory location where the first instruction of the subroutine is stored. We focus on graphics subroutines, subroutines that read the keyboard, and subroutines that let you print the contents of the accumulator. A few new machine language commands and a new addressing mode also are introduced.

### **X and Y Registers**

Apple's built-in subroutines require certain data if they are to work. For example, if you wish to plot a point in graphics mode, you need to tell the plotting routine the X and Y coordinates of the point you wish to plot. Apple's subroutines expect to find the required data in certain special locations. These locations may be particular numbered memory locations or the accumulator. Sometimes the X or Y register must be loaded with the data. These registers are very similar to the accumulator. Like the accumulator, they are memory locations that are referred to by a name rather than by a number. They behave as ordinary locations do, except they are also used for a few special purposes. In addition to being the place where data for some subroutines must be stored, the registers are extremely convenient for use as counters. When machine language loops are described, we make extensive use of the X and Y registers for counting. They are also used as indexes in the absolute indexed addressing mode. This addressing mode is described in detail later in this chapter.

### Plotting Points

The statements we used to plot points in BASIC have machine language equivalents. By using the built-in subroutines, we are able to translate a BASIC program that plots a point directly into machine language. Consider the following BASIC program.

```
10 GR
20 COLOR= 15
30 ROW= 5
40 COL= 10
50 PLOT COL, ROW
```

Note that the X coordinate, which controls the column in which the point is printed, comes first in the PLOT statement.

From the above program, we see that the data required for the PLOT routine to work are a COLOR, a row (Y coordinate), and a column (X coordinate) for the point. In machine language, we store the data in the following locations.

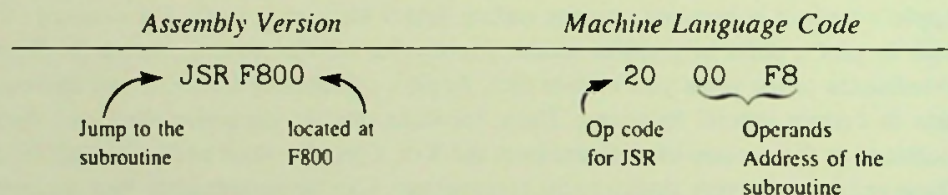
Color: Store in location \$30.

Row: Store in the accumulator.

Column: Store in the Y register.

Once the data are stored properly, we can use a special plotting routine to plot the point. Colors are given the same numerical codes in machine language as they had in BASIC. (See Appendix C for a list of colors and their corresponding numerical codes.) The color numbers must be converted into HEX, of course. Each of the colors will thus be coded 0 to F. When we store a color value, we put its code into each nibble of location \$30. That is, color number 1 is stored as \$11, color number 2 is stored as \$22, color number 3 is stored as \$33, and so on.

The plotting routine is located beginning in location \$F800. To use the Plot a Point subroutine, jump to this location using the machine language command:



Thus JSR is analogous to GOSUB in BASIC.

The following table is included for the reader's convenience. It summarizes the information that the PLOT subroutine requires and the locations in which this information must be stored.



*PLOT Subroutine: JSR F800*

| <i>Color</i> | <i>Column</i> | <i>Row</i>  |
|--------------|---------------|-------------|
| 0030         | Y Register    | Accumulator |

Before the Plot a Point routine actually can be used, you first must enter the graphics mode. This is done by using another special subroutine, located at FB40.

| <i>Assembly Version</i> | <i>Machine Language Code</i> | <i>Function</i>                |
|-------------------------|------------------------------|--------------------------------|
| JSR FB40                | 20 40 FB                     | Used to set the Graphics mode. |

The following program uses these subroutines to plot a point in row 5, column 10 (decimal).

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Equivalent BASIC Program</i> |
|--------------------------|-------------------------|---------------------------------|
| JSR FB40                 | 20 40 FB                | GR                              |
| LDA #FF                  | A9 FF                   | COLOR = 15                      |
| STA \$0030               | 8D 30 00                |                                 |
| LDY #0A                  | A0 0A                   | COL = 10                        |
| LDA #05                  | A9 05                   | ROW = 5                         |
| JSR F800                 | 20 00 F8                | PLOT COL, ROW                   |
| RTS                      | 60                      | END                             |

The machine language program could be entered using the monitor as shown here.

```
* 300:20 40 FB A9 FF 8D 30 00
* 308:A0 0A A9 05 20 00 F8 60
```

Note the new command LDY (*LoaD the Y register*, op code A0). This command works exactly as the LDA command encountered earlier does. In this example, LDY is being used in the immediate addressing mode. As was the case with LDA, the command can also be used in the absolute addressing mode. Analogous commands also exist for loading the X register (LDX). The op codes for all these commands can be found in the table in Appendix B.

## Zero-Page Memory

Addresses are four-digit HEX numbers. The first two digits (as written on paper, not as typed into the monitor) are the nibbles of the most significant byte (MSB) of the address. This byte is also referred to as the *page* of the address. For example, location 0350 is in page

03 of memory, and Address C030 is in page C0 of memory. In the Plot a Point program of the last section, the command STA 0030 stores the code FF (for white, color number fifteen) into the *zero-page* address 0030. If the computer knew you were referring to a zero-page location, then 30 could be used to refer to the address instead of 0030. In fact, you can refer to a zero-page location using a two-digit HEX number if you use machine language instructions in the zero-page addressing mode. Many commands, such as LDA, STA, and LDX, can be used in the zero-page addressing mode. When the special zero-page mode op codes are used, the only operand that need be given is the least significant byte of the zero-page address. For example, each of these alternatives successfully stores \$FF in location \$30:

|                                 | <i>Assembly Language</i> | <i>Machine Language</i> |
|---------------------------------|--------------------------|-------------------------|
| Absolute<br>addressing<br>mode  | LDA #FF                  | A9 FF                   |
|                                 | STA \$0030               | 8D 30 00                |
| Zero-page<br>addressing<br>mode | LDA #FF                  | A9 FF                   |
|                                 | STA \$30                 | 85 30                   |

In the zero-page mode, the operand of STA is only 1 byte long. This is the only advantage of the zero-page addressing mode. Since zero-page commands are only 2 bytes long, a program using them would execute slightly faster than one using the absolute addressing mode, whose commands are 3 bytes long. There is no real conceptual difference between these two addressing modes, however. Both STA \$0030 and STA \$30 store the accumulator's contents in a memory location that is referred to by number. In fact, the mini-assembler program (which automatically converts the mnemonic codes of assembly language into the op codes and operands of machine language) will use op code 85 for STA, whether STA \$0030 or STA \$30 is used. Since both the absolute addressing mode and zero-page addressing mode really accomplish the same thing, the mini-assembler will choose the more efficient alternative for addresses less than \$100.

### Addresses of Machine Language Subroutines

Suppose you are working in the graphics mode and your machine language program is not functioning properly. To debug the program, you need to get a listing first, but in graphics mode there are only four lines of text on the bottom of the screen. At this point, it would be useful to know the machine language equivalent of the BASIC command TEXT. The addresses for all useful Apple subroutines are given in Appendix D. Some may be interested in learning how to figure out the address of a subroutine, because knowing how to do so could be valuable if you needed to use a subroutine whose address is not given in Appendix



D. Much information can be gleaned from the ROM listing given on pages 136–171 of the *Apple II Reference Manual*. On page 162, you can look up the address of the Set Graphics command (FB40). Note that SETGR for SET GRaphics appears here. Since related subroutines are located close to one another in many cases, you might guess that the Set Text subroutine would be nearby. Sure enough, location FB39 has the comment SETTXT in the ROM listing. This means the statement

| <i>Assembly Language</i> | <i>Machine Language</i> |
|--------------------------|-------------------------|
| JSR FB39                 | 20 39 FB                |

is equivalent to TEXT in BASIC. You could type

```
*800:20 39 FB 60
```

to enter a program whose only function is to set the TEXT mode. By using the monitor commands

```
*800G
*300L
```

you could conveniently get a listing of your graphics program (assuming you stored it starting in location \$300). When the program is debugged, you could use

```
*300G
```

to run it. Notice that two programs can be stored in Apple's memory. Either one can be run by selecting the appropriate address in the monitor GO command.

Finally, it is easy to find machine language equivalents for POKE and CALL statements of BASIC. You have to convert the decimal address given in the POKE or CALL statement to HEX as a first step. If you use this HEX address in conjunction with a STA statement, you have the machine language equivalent of the POKE statement since all POKE does is store data in Apple's memory. Also the JSR command is the machine language equivalent of CALL, since all CALL does is execute a machine language subroutine from within a BASIC program. Therefore the converted decimal address used in conjunction with JSR is the machine language equivalent of the BASIC CALL statement.

### Other Graphics Subroutines

In BASIC, HLIN A,B AT C draws a horizontal line starting in column A and ending in column B. The line is drawn in row C. The machine language equivalent of HLIN is the

statement JSR F819. As with the Plot a Point routine, certain information must be stored in special locations before the Plot a Horizontal Line subroutine can be used. The following table summarizes the information that the HLIN routine requires and the locations in which the data must be stored.

*HLIN subroutine: JSR F819*

| <i>Color</i> | <i>Starting Column</i> | <i>Ending Column</i> | <i>Row</i>  |
|--------------|------------------------|----------------------|-------------|
| 0030         | Y register             | 002C                 | Accumulator |

**Example**

The following program would plot a horizontal line in row \$12 that begins in column \$07 and ends in column \$21.

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>BASIC</i>  | <i>Comments</i>                                          |
|--------------------------|-------------------------|---------------|----------------------------------------------------------|
| JSR FB40                 | 20 40 FB                | GR            | Set Graphics mode.                                       |
| LDA #BB                  | A9 BB                   | COLOR = 11    | Load a color for the line (eleven used in this program). |
| STA 0030                 | 8D 30 00                |               |                                                          |
| LDY #07                  | A0 07                   | A = 7         | Starting column is seven.                                |
| LDA #21                  | A9 21                   | B = 33        | Ending column is thirty three.                           |
| STA \$2C                 | 85 2C                   |               |                                                          |
| LDA #12                  | A9 12                   | C = 18        | Line is to be in row eighteen.                           |
| JSR F819                 | 20 19 F8                | HLIN A,B AT C | Plot the line.                                           |
| RTS                      | 60                      | END           |                                                          |

Similarly the statement VLIN A, B AT C plots a vertical line in column C starting in row A and ending in row B (in BASIC). The machine language equivalent of VLIN is the subroutine located at location F828. The following table summarizes the data the VLIN routine requires and the locations in which the data must be stored.

*VLIN subroutine: JSR F828*

| <i>Color</i> | <i>Starting Row</i> | <i>Ending Row</i> | <i>Column</i> |
|--------------|---------------------|-------------------|---------------|
| 0030         | Accumulator         | 002D              | Y register    |

**Example**

The following program would plot a vertical line in column \$11 that begins in row \$02 and ends in row \$15.



| <i>Assembly Language</i> | <i>Machine Language</i> | <i>BASIC</i>  | <i>Comments</i>                                |
|--------------------------|-------------------------|---------------|------------------------------------------------|
| JSR FB40                 | 20 40 FB                | GR            | Set Graphics mode.                             |
| LDA #BB                  | A9 BB                   | COLOR = 11    | Use color number eleven.                       |
| STA \$30                 | 85 30                   |               |                                                |
| LDA #15                  | A9 15                   | B = 21        | End the line in row twenty-one.                |
| STA \$2D                 | 85 2D                   |               |                                                |
| LDA #02                  | A9 02                   | A = 2         | Start the line in row 2.                       |
| LDY #11                  | A0 11                   | C = 17        | The line is to be graphed in column seventeen. |
| JSR F828                 | 20 28 F8                | VLIN A,B AT C | Plot the line.                                 |
| RTS                      | 60                      | END           |                                                |

### Effect of Subroutines on the Accumulator, X Register, and Y Register

Consider this assembly listing designed to plot two points—one in row \$01, column \$05, and the other in row \$01, column \$10.

| <i>Assembly Language</i> | <i>Comments</i>                        |
|--------------------------|----------------------------------------|
| JSR FB40                 | Set Graphics mode.                     |
| LDA #FF                  | Store the color.                       |
| STA \$0030               |                                        |
| LDY #05                  | Load column of first point.            |
| LDA #01                  | Load row of first point.               |
| JSR F800                 | Plot first point.                      |
| LDY #10                  | Change the column for the second point |
| JSR F800                 | and plot the second point.             |
| RTS                      |                                        |

Logically all seems in order here. The row and column of the first point are loaded and the first point is then plotted. The column of the second point is different from that of the first point, but the row is the same. Therefore, prior to plotting the second point, only the column number was loaded (Y was loaded with \$10). The assumption here is that the accumulator will still contain the row number \$01. This is *not* so, because the use of built-in subroutines frequently scrambles the contents of the accumulator, X register, or Y register. Therefore, after the first JSR F800 command, we *cannot* assume that the accumulator still contains the value \$01. In fact, if the foregoing program is run, the results will not consist of points in row 1, column 5, and row 1, column 10. The correct program for plotting these two points would reload the accumulator with the row value after plotting the first point.

| <i>Assembly Version</i> | <i>Comments</i>               |
|-------------------------|-------------------------------|
| JSR FB40                | Set Graphics mode.            |
| LDA #FF                 | Store the                     |
| STA \$30                | color number.                 |
| LDY #05                 | Store the column              |
| LDA #01                 | and row for the first point.  |
| JSR F800                | Plot the first point.         |
| LDY #10                 | Store the column              |
| LDA #01                 | and row for the second point. |
| JSR F800                | Plot the second point.        |
| RTS                     |                               |

Notice that the color was not reloaded prior to plotting the second point. This is so because the color was not stored in the accumulator, X register, or Y register. In general, built-in subroutines will affect only the accumulator and registers. (Although some subroutines *do* affect other memory locations. For example, JSR FB40 will erase any color value stored in \$30.) The lesson to be learned here may be summarized by saying that whenever any built-in subroutine is used, make sure you reload the accumulator, X register, and Y register, if the values stored in these locations prior to using the subroutine need to be used again later.

It should be mentioned that not all built-in subroutines will scramble the contents of the accumulator, the X register, or the Y register. However, many built-in subroutines will scramble the contents of one or more of these locations. Rather than trying to remember which subroutines scramble which locations, it is simpler just to make a practice of reloading all three locations whenever a built-in subroutine is used.

## ASCII Codes

In machine language, the Apple handles all character manipulations through the use of a special numerical code. Each keyboard character (and some characters that cannot be accessed through the keyboard, such as the [ character) has a number associated with it. The American Standard Code for Information Interchange (ASCII) is used by the Apple. A list of the ASCII codes for the various characters is given in Appendix E. A few points are worthy of note. First, the letter A has ASCII code \$C1 and the rest of the letters' codes are in ascending order. Thus to find the ASCII code of a letter without using a table, just add the letter's position in the alphabet to \$C0. For example, since S is the 19th letter in the alphabet, its ASCII code is

$$\$C0 + \$13 = \$D3$$

The digits 0 through 9 have ASCII codes \$B0 through \$B9. Two other ASCII codes you will



probably use frequently are \$A0 to represent a blank space and \$8D to represent a carriage return.

### Printing Characters in Machine Language

There is a built-in subroutine located at \$FDED that prints the character whose ASCII code is in the accumulator. After printing the character, the cursor position is automatically advanced (a new line is begun after the cursor's position is advanced past the last position on the line). Unlike PRINT in BASIC, the machine language subroutine will not print output on a new line each time it is executed. To move the cursor to a new line, you must load the code \$8D (for a carriage return) into the accumulator and "print" it using JSR FDED. In machine language, characters are printed one at a time, so that printing information using the machine language subroutine FDED is particularly tedious. Consider the following program.

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>     |
|--------------------------|-------------------------|---------------------|
| LDA #C1                  | A9 C1                   | Print A.            |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #BD                  | A9 BD                   | Print =.            |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #A0                  | A9 A0                   | Print a space.      |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #B1                  | A9 B1                   | Print 1.            |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #8D                  | A9 8D                   | Skip to a new line. |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #C2                  | A9 C2                   | Print B.            |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #BD                  | A9 BD                   | Print =.            |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #B2                  | A9 B2                   | Print 2.            |
| JSR FDED                 | 20 ED FD                |                     |
| LDA #B3                  | A9 B3                   | Print 3.            |
| JSR FDED                 | 20 ED FD                |                     |
| RTS                      | 60                      | End.                |

The output is

A= 1  
B=23

Clearly machine language is not designed for printing fancy labels with your output. If it does become necessary to print a long character string in machine language, the most efficient way to do this is to load the ASCII codes for each character manually, using the monitor. Place the ASCII codes in consecutive memory locations. You can then use a machine language loop (these are described in the next chapter) together with absolute indexed addressing (described shortly) to "travel through" the list of ASCII codes and print each character.

### Printing Numerical Information

Many times, the result that needs to be printed will be a number. Notice that the subroutine located at FDED does not print the actual number stored in the accumulator. Instead it prints the character whose ASCII code is stored in the accumulator. There is a subroutine that prints the numerical contents of the accumulator rather than the character whose ASCII code is stored in the accumulator.

This subroutine is located at \$FDDA. Like the subroutine located at \$FDED, output will not be on separate lines unless you "print" a carriage return between output items. Consider the following example.

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                                                            |
|--------------------------|-------------------------|--------------------------------------------------------------------------------------------|
| LDA 390                  | AD 90 03                | Print the numerical contents of location 390.                                              |
| JSR FDDA                 | 20 DA FD                |                                                                                            |
| LDA #8D                  | A9 8D                   | Skip to a new line. Note that JSR FDED is used to print the carriage return, not JSR FDDA. |
| JSR FDED                 | 20 ED FD                |                                                                                            |
| LDA 391                  | AD 91 03                | Print the numerical contents of location 391.                                              |
| JSR FDDA                 | 20 DA FD                |                                                                                            |
| RTS                      | 60                      | End.                                                                                       |

The subroutine located at \$FDDA always outputs two-digit HEX numbers. Thus, if 01 were stored in \$390 and \$12 stored in \$391, the output of the above program would be

01  
12

not

1  
12



The subroutine SFDDA scrambles the contents of the accumulator, so remember to store the contents of the accumulator in a memory location, if you will need them after they are printed. Consider the following program segments.

| <i>Assembly Language</i> | <i>Comments</i>                                |
|--------------------------|------------------------------------------------|
| .                        |                                                |
| .                        |                                                |
| .                        |                                                |
| LDA #01                  | The contents of the accumulator are stored for |
| STA 390                  | safekeeping.                                   |
| JSR FDDA                 | Print the contents of the accumulator.         |
| LDA 390                  | The value 01 is available again in the         |
| .                        | accumulator.                                   |
| .                        |                                                |
| .                        |                                                |

| <i>Assembly Language</i> | <i>Comments</i>                                   |
|--------------------------|---------------------------------------------------|
| .                        | In this version, 01 is printed as before, but the |
| .                        | value 01 is no longer stored in the accumu-       |
| .                        | lator.                                            |
| LDA #01                  |                                                   |
| JSR FDDA                 |                                                   |
| .                        |                                                   |
| .                        |                                                   |
| .                        |                                                   |

### Input from the Keyboard

For most applications, it is easier to load data into memory manually using such monitor commands as

\*390:57 82 4F

than it is to use the machine language equivalent of INPUT. However, there are times when you may want to input an item or two from the keyboard during a program. The INPUT subroutine in machine language is located at \$FD35. Actually this routine is more closely related to GET than it is to INPUT. When JSR FD35 is encountered, program execution is halted and a flashing cursor is displayed. The ASCII code for the first key hit is stored in the

accumulator and execution resumes. You can input only one ASCII code per JSR FD35 command, just as with GET in BASIC. Two points need to be stressed here. First, when you enter information from the keyboard using INPUT in BASIC, the keys you strike are displayed on the screen. This is not true in machine language. If the subroutine \$FD35 is used, you will be prompted with a flashing cursor, but nothing else will be displayed unless you use the subroutine \$FDED to display the character after it is read in. Second, remember that the ASCII code for the keystroke is stored in the accumulator by the \$FD35 subroutine, not the actual number or letter shown on the keyboard key. For example, in the program segment

#### Assembly Language

```
JSR FD35  
JSR FDDA
```

if the "1" key is hit, the number \$B1 will be stored in the accumulator and \$B1 will be displayed on the screen.

An alternate subroutine located at \$FD1B also reads the keyboard. It does not issue a flashing cursor as a prompt and does not recognize escape codes as the subroutine \$FD35 does. In all other respects, subroutines \$FD1B and \$FD35 are identical.

### Random Numbers

The Read the Keyboard subroutines located at \$FD1B and \$FD35 can be used to generate random numbers. While these routines are waiting for you to hit a key, they continually add \$01 to a four-digit HEX number stored in locations \$4E and \$4F. The addition stops when you hit a key. Since the exact time in milliseconds it takes you to strike a key is essentially random, the final values stored in \$4E and \$4F will be quite unpredictable.

One drawback to this method of random number generation is the fact that it depends on the user actually striking a key. You get one four-digit random number (or two two-digit random numbers) each time you hit a key. Suppose the following commands are put in a machine language loop.



| <i>Assembly Language</i>  | <i>Machine Language</i> | <i>Comments</i>                                                                                                                                                                                                                                                                                                                |
|---------------------------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JSR FD35                  | 20 35 FD                | The ASCII code loaded into the accumulator by the first command is destroyed by the second command. This is done because we are interested in the random number generated by JSR FD35, not the actual keystroke used to generate the number. Also notice that LDA is being used in the zero-page addressing mode (op code A5). |
| LDA \$4E                  | A5 4E                   |                                                                                                                                                                                                                                                                                                                                |
| JSR FDDA <i>actual #</i>  | 20 DA FD                | Print the first random number and skip to a new line.                                                                                                                                                                                                                                                                          |
| LDA #8D <i>CR</i>         | A9 8D                   |                                                                                                                                                                                                                                                                                                                                |
| JSR FDED <i>character</i> | 20 ED FD                | Print the second random number.                                                                                                                                                                                                                                                                                                |
| LDA \$4F                  | A5 4F                   |                                                                                                                                                                                                                                                                                                                                |
| JSR FDDA <i>actual #</i>  | 20 DA FD                | These commands are included so that when the instructions shown here are executed again, the next random number printed will be on a new line.                                                                                                                                                                                 |
| LDA #8D <i>CR</i>         | A9 8D                   |                                                                                                                                                                                                                                                                                                                                |
| JSR FDED <i>character</i> | 20 ED FD                |                                                                                                                                                                                                                                                                                                                                |

If you wish to print, say, 100 two-digit random numbers, you are going to have to hit 50 keys. Program execution will be stopped each time the Read the Keyboard subroutine waits for your keystroke.

### The AND Command

Suppose A and B represent statements that are either true or false. Then the expression "A AND B" is true if and only if both A and B are true. We can construct a truth table showing the value of the expression "A AND B" for all possible combinations of values for A and B. In "computerese," 0 stands for false and 1 stands for true. Hence our truth table becomes

| <i>A</i> | <i>B</i> | <i>Value of "A AND B"</i> |
|----------|----------|---------------------------|
| 1        | 1        | 1                         |
| 1        | 0        | 0                         |
| 0        | 1        | 0                         |
| 0        | 0        | 0                         |

Thus, unless both A and B are ones, the value of "A AND B" will always be zero.

Let us examine what it means to AND two HEX numbers together. Conceptually this is done by ANDing each pair of corresponding bits in the two numbers. All eight pairs of bits are processed to find the 8-bit result. For example, let us find A9 AND 3F. First represent each HEX number in binary.

A9 = 10101001      and      3F = 00111111

Next write the binary representation for one HEX number under the representation of the other

|           |         |   |   |   |   |         |   |   |   |   |   |   |
|-----------|---------|---|---|---|---|---------|---|---|---|---|---|---|
| A9=       | 1       | 0 | 1 | 1 | 0 | 1       | 1 | 0 | 1 | 0 | 1 | 1 |
| 3F=       | 0       | 1 | 0 | 1 | 1 | 1       | 1 | 1 | 1 | 1 | 1 | 1 |
| <hr/>     |         |   |   |   |   |         |   |   |   |   |   |   |
| A9 AND 3F | 0       | 1 | 0 | 1 | 1 | 0       | 1 | 1 | 0 | 1 | 0 | 1 |
|           | ↑       |   |   |   |   | ↑       |   |   |   |   |   |   |
|           | since   |   |   |   |   | since   |   |   |   |   |   |   |
|           | 1 AND 0 |   |   |   |   | 1 AND 1 |   |   |   |   |   |   |
|           | is 0    |   |   |   |   | is 1    |   |   |   |   |   |   |

Work with vertical pairs of bits, from left to right. All bits in the answer will be zeros except those that correspond to a vertical column containing two ones. Also notice that any bit ANDed with one is left untouched: if you AND a *zero* bit with a one bit, the result is a *zero* bit. If you AND a *one* bit with a one bit, the result is a *one* bit. In our example, the last 6 bits in the answer are the same as the last 6 bits of the top number (A9) because the last 6 bits of the bottom number (3F) are all ones. We learn that the result of ANDing A9 with 3F is

00101001 or \$29

Now that you can "AND" two numbers on paper, we can discuss how the AND command works in machine language. We will examine two addressing modes—the immediate addressing mode and the absolute addressing mode. In the immediate addressing mode, the command

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comment</i>                                              |
|--------------------------|-------------------------|-------------------------------------------------------------|
| AND #3F                  | 29 3F                   | The op code for AND in the immediate addressing mode is 29. |

will take whatever is stored in the accumulator and "AND" this number with \$3F. The result is stored back in the accumulator, wiping out the number originally stored there. In absolute addressing mode, the command



| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comment</i>                                             |
|--------------------------|-------------------------|------------------------------------------------------------|
| AND 390                  | 2D 90 03                | The op code for AND in the absolute addressing mode is 2D. |

will AND the byte in the accumulator with the contents of location \$390. The result is stored back in the accumulator, wiping out the number originally stored there.

### Using the AND Command with the Get a Keystroke Subroutine

We now look at one application of the AND command. Suppose you want to plot a point on the screen whose row and column are entered from the keyboard, instead of using a command such as LDA #12 to load the row of the point into the accumulator. An obvious disadvantage of using LDA #12 to load row \$12 into the accumulator is that your program is not general; it will always plot the point in row \$12. Consider this first attempt to enter the point's position from the keyboard. In this program, the row and column number of the point are supposed to be set to the same value.

| <i>Assembly Language</i> | <i>Comments</i>                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JSR FB40                 | Set Graphics mode.                                                                                                                                                                                                                                                                                                                                                                                          |
| LDA #FF                  | Set the color.                                                                                                                                                                                                                                                                                                                                                                                              |
| STA 0030                 |                                                                                                                                                                                                                                                                                                                                                                                                             |
| JSR FD35                 | Read the keyboard.                                                                                                                                                                                                                                                                                                                                                                                          |
| TAY                      | This new command Transfers data from the Accumulator to the Y register. Remember, FD35 reads the keyboard and puts the ASCII code of the key you hit into the accumulator. To get the column number into the Y register, TAY (op code A8) was used. After executing TAY, the number read from the keyboard still remains in the accumulator. The row and column numbers of the point will thus be the same. |
| JSR F800                 | Plot the Point.                                                                                                                                                                                                                                                                                                                                                                                             |
| RTS                      |                                                                                                                                                                                                                                                                                                                                                                                                             |

One of the problems with this program is that ASCII codes are stored in the accumulator by the subroutine \$FD35. Suppose we hit the "2" key because we want our point plotted in row 2, column 2. The accumulator will *not* contain the byte 02, but the ASCII code representing the "2" key (which is B2) instead. Thus the program will *not* work. To correct

this problem, we need to get rid of the first nibble in \$B2. This can be done by using the command

```
AND #0F      ($0F= 00001111 in binary)
```

after reading the keyboard. Since the left 4 bits of the accumulator are ANDed with zeros, the result will have the first 4 bits all zeros (eliminating the left nibble). The right 4 bits of the original number will be left untouched, as they are being ANDed with ones. Therefore the command AND#0F converts the ASC code for a digit (from 0 to 9) into the numerical value of the digit.

The following revised program will let us enter row and column numbers for the point from the keyboard.

| <i>Assembly Version</i> | <i>Machine Language Version</i> |
|-------------------------|---------------------------------|
| JSR FB40                | 20 40 FB                        |
| LDA #FF                 | A9 FF                           |
| STA \$30                | 85 30                           |
| JSR FD35                | 20 35 FD                        |
| AND #0F                 | 29 0F                           |
| TAY                     | A8                              |
| JSR F800                | 20 00 F8                        |
| RTS                     | 60                              |

A few comments need to be made about this program. First, the row and column numbers cannot be higher than \$F. Remember that the Read the Keyboard subroutine located at FD35 enables you to get just one ASCII code from the keyboard. Reading two-digit numbers will require the use of two JSR FD35 commands. The two separate digits will then have to be combined into a single two-digit number. Techniques for doing this are described in Chapter 6.

Second, you should note that TAY is a (1-byte-long) implied addressing mode command. The addresses of the two locations involved when the TAY command is executed do not have to be stated as operands—they are *implied* in the command's op code. A similar command also exists to handle the flow of information from the Y register to the accumulator. Analogous commands exist for the X register. In summary,

```
TAY  Transfer data from the Accumulator to the Y register.
TYA  Transfer data from the Y register to the Accumulator.
TAX  Transfer data from the Accumulator to the X register.
TXA  Transfer data from the X register to the Accumulator.
```

The op codes for these commands can be found in Appendix B.



**NORMAL, FLASH, and INVERSE in Machine Language**

In machine language, each keyboard character has three different ASCII codes associated with it—one for NORMAL printing (white letters on a black background), one for INVERSE printing (black letters on a white background), and one for FLASHing print (characters displayed alternately in NORMAL and INVERSE print). The ASCII codes for each of the three types of printing are given for each keyboard character in Appendix E.

We have seen that AND can be used to convert ASCII codes for digits into the numerical value of the digits. Another application of the AND command is given here. We will show how to convert ASCII codes for NORMAL print into the ASCII codes for INVERSE and FLASHing print.

The following two conversion rules can be used on any letter of the alphabet. (Rule 1 works only on characters whose ASCII codes are greater than or equal to \$C0. Therefore rule 1 would not work on the digits 1 through 9. Rule 2 can be used to convert the NORMAL ASCII codes for the digits into INVERSE ASCII codes, however.)

1. The ASCII code for a FLASHing character is found by setting the leftmost bit in the NORMAL ASCII code equal to zero.
2. The ASCII code for an INVERSE character is found by setting the 2 leftmost bits in the NORMAL ASCII code equal to zeros.

For example, let us use these rules to find the codes needed to display the letter A in FLASHing and INVERSE modes. The ASCII code for the NORMAL letter A is \$C1. In binary,

C1 = 11000001

Therefore, to print a FLASHing A, use the ASCII code

01000001 or \$41

↑  
left bit set equal to zero

and to print an INVERSE A, use the ASCII code

00000001 or \$01

2 left bits set equal to zero

You should see that the command AND #7F will convert a NORMAL ASCII code stored in the accumulator to the FLASHing code. Note that \$7F equals 01111111 in binary.

When the code stored in the accumulator is ANDed with 7F, its leftmost bit is set to zero, because the leftmost bit of 7F is 0. The other 7 bits in the code are left untouched because they are being ANDed with ones.

Similarly use the command AND #3F to convert a NORMAL ASCII code stored in the accumulator to the INVERSE code. In binary,

\$3F= 00111111

Therefore, when the code stored in the accumulator is ANDed with 3F, its 2 leftmost bits will both be set to zero. The other bits in the code will be left untouched, since they are being ANDed with one bits.

Rule 1 will not work on characters whose ASCII codes are less than \$C0. To see why, let us attempt to apply the rules to find the FLASHing and INVERSE codes for the asterisk. The NORMAL ASCII code for the asterisk is \$AA. In binary,

\$AA = 10101010

so according to rule 2 the INVERSE ASCII code must be

00101010      or \$2A

two leftmost bits  
set equal to zero

In fact, \$2A is the correct code for an INVERSE asterisk. According to rule 1, the FLASHing ASCII code must be

00101010      or \$2A  
↖  
leftmost bit set  
equal to zero

Obviously the FLASHing code for an asterisk cannot be the same as the INVERSE code. The correct code for the FLASHing asterisk is

01101010      or \$6A



In general, the second bit from the left is equal to one in all FLASHing codes, even if this bit is equal to zero in the NORMAL ASCII code. Therefore, to find the FLASHing code for any character, you apply rule 1 and change the second bit from the left to a one if it is a zero. It turns out that there are machine language commands that will convert any normal ASCII code to a FLASHing code. As the need to do so arises infrequently, this is not discussed in detail here. For now, if you need to print a character whose ASCII code is less than \$C0 in FLASHing mode, just look up the FLASHing code in Appendix E and load the accumulator in immediate or absolute addressing mode with the proper code.

We close this section by giving a complete example machine language program that uses many of the commands and techniques introduced so far. For any key that you strike, the program prints:

the ASCII code of the key and the corresponding character in NORMAL print on the first line,  
the ASCII code and the corresponding character in INVERSE print on the second line,  
and  
the ASCII code and the corresponding character in FLASHing print on the third line  
(this part does not work if the NORMAL ASCII code is less than \$C0).

The monitor commands that would be used to enter the program are shown here. A complete flowchart and assembly listing are given on the next page.

```
*300:20 35 FD 8D 90 03 20 DA
*308:FD A9 A0 20 ED FD AD 90
*310:03 20 ED FD A9 8D 20 ED
*318:FD AD 90 03 29 3F 8D 91
*320:03 20 DA FD A9 A0 20 ED
*328:FD AD 91 03 20 ED FD A9
*330:8D 20 ED FD AD 90 03 29
*338:7F 8D 91 03 20 DA FD A9
*340:A0 20 ED FD AD 91 03 20
*348:ED FD 60
```

| <i>Flowchart</i>                                   | <i>Assembly<br/>Language</i> | <i>Machine<br/>Language</i> | <i>Comments</i>                                                                                                                                                                                                                                              |
|----------------------------------------------------|------------------------------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Read keyboard                                      | JSR FD35                     | 20 35 FD                    | The ASCII code is stored in 390 so that it will be available later in the program.                                                                                                                                                                           |
| Display ASCII code                                 | STA 390                      | 8D 90 03                    |                                                                                                                                                                                                                                                              |
| for the keystroke                                  | JSR FDDA                     | 20 DA FD                    |                                                                                                                                                                                                                                                              |
| Print a space                                      | LDA #A0                      | A9 A0                       |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Display character in                               | LDA 390                      | AD 90 03                    |                                                                                                                                                                                                                                                              |
| NORMAL print                                       | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Skip to a new line                                 | LDA #8D                      | A9 8D                       |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Calculate and display the INVERSE mode ASCII code  | LDA 390                      | AD 90 03                    | The ASCII code for the INVERSE character is needed for two purposes. The code itself needs to be printed, then the INVERSE character represented by the code must be printed. The ASCII code is saved in 391 so that it will be available for both purposes. |
|                                                    | AND #3F                      | 29 3F                       |                                                                                                                                                                                                                                                              |
|                                                    | STA 391                      | 8D 91 03                    |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDDA                     | 20 DA FD                    |                                                                                                                                                                                                                                                              |
| Print a space                                      | LDA #A0                      | A9 A0                       |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Display character in                               | LDA 391                      | AD 91 03                    |                                                                                                                                                                                                                                                              |
| INVERSE print                                      | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Skip to a new line                                 | LDA #8D                      | A9 8D                       |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Calculate and display the FLASHing mode ASCII code | LDA 390                      | AD 90 03                    | The FLASHing code is saved in 391 for the same reason the INVERSE ASCII code had to be saved.                                                                                                                                                                |
|                                                    | AND #7F                      | 29 7F                       |                                                                                                                                                                                                                                                              |
|                                                    | STA 391                      | 8D 91 03                    |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDDA                     | 20 DA FD                    |                                                                                                                                                                                                                                                              |
| Print a space                                      | LDA #A0                      | A9 A0                       |                                                                                                                                                                                                                                                              |
|                                                    | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| Display character in                               | LDA 391                      | AD 91 03                    |                                                                                                                                                                                                                                                              |
| FLASHing print                                     | JSR FDED                     | 20 ED FD                    |                                                                                                                                                                                                                                                              |
| End                                                | RTS                          | 60                          |                                                                                                                                                                                                                                                              |



### An Alternate Method of Displaying Text

Appendix F contains a “map” of the text screen. The text screen consists of 24 rows and 40 columns. Each of the columns is labeled with a HEX number from \$00 to \$27. Each row is labeled with a three-digit HEX number. These numbers do not follow any particular orderly pattern. This map is used to find the memory location corresponding to any square on the text screen grid. To find the memory location corresponding to any square of the text screen grid, add the HEX address found at the beginning of the square’s row to the HEX number of the square’s column. For example, the memory location corresponding to row 18, column 25, is

$$\begin{array}{c} \$550 + \$19 = \$569 \\ \uparrow \qquad \qquad \uparrow \\ \text{HEX number} \qquad \text{Column} \\ \text{corresponding} \qquad \text{25, ($19)} \\ \text{to row 18,} \\ \text{($12)} \end{array}$$

Note that the first column is labeled \$00 so that column 25 is actually the 26th column from the left side of the screen. Similarly row 18 is really the 19th row from the top of the screen.

To display any character on the text screen, simply store the ASCII code for the character in the memory location corresponding to the row and column where you want the character to appear. For example, the program

| <i>Assembly Language</i> | <i>Machine Language</i> |
|--------------------------|-------------------------|
| LDA #C1                  | A9 C1                   |
| STA 0569                 | 8D 69 05                |
| RTS                      | 60                      |

would print the letter A (ASCII code C1) in row \$12, column \$19.

### Absolute Indexed Addressing Mode

In BASIC, we can think of a subscripted list as a list of data contained in consecutive memory locations. Schematically list X might look like

|          |      |      |      |      |      |     |      |
|----------|------|------|------|------|------|-----|------|
| Contents | 79   | 42   | 38   | 45   | 22   | ... | 8    |
| Location | X(0) | X(1) | X(2) | X(3) | X(4) | ... | X(Y) |

The subscripts in parentheses tell you how far to the right to travel in the list X to find the element of interest. For example, subscript 2 can be interpreted to mean "go two places to the right of X(0) to find the storage location, which (in this case) contains 38."

In machine language, we have an analogous situation. Instead of giving our lists names, we give a *base address* and use an index (either the X or Y register) to act as a subscript. Consider the following list represented schematically.

|          |          |          |          |          |          |     |         |
|----------|----------|----------|----------|----------|----------|-----|---------|
| Contents | 79       | 42       | 38       | 45       | 22       | ... | 8       |
| Location | 400,0    | 400,1    | 400,2    | 400,3    | 400,4    | ... | 400,Y   |
|          | 400 + 0  | 400 + 1  | 400 + 2  | 400 + 3  | 400 + 4  |     | 400 + Y |
|          | or \$400 | or \$401 | or \$402 | or \$403 | or \$404 |     |         |

The commands

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                                                                                                                                                            |
|--------------------------|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDY #02                  | A0 02                   | B9 is the op code for Load the Accumulator in the absolute indexed addressing mode. The assembly language shorthand 400,Y represents the indexed list with base address \$400 and index Y. |
| LDA 400,Y                | B9 00 04                |                                                                                                                                                                                            |

would load the accumulator with 38. The 400 is a base address and since  $Y = 02$ , we go two addresses to the right of 400 to find the address from which to load. In other words, LDA 400,Y loads the accumulator from location  $\$400 + \text{Value stored in Y register} (= 400 + 02 = \$402, \text{ in our case})$ . You should see that absolute indexed addressing enables you to mimic subscripted variables of BASIC. This addressing mode is ideal for working with data that are stored (or are to be stored) in consecutive memory locations. Many of the commands with which you are familiar can be used in the absolute indexed addressing mode, such as LDA, STA, and AND. The X register can be used as an index just as the Y register (LDA 400,Y and LDA 400,X both do essentially the same thing, but each command has a different op code). Absolute indexed addressing mode commands are 3 bytes long. The first byte is an op code that selects the operation (LDA, etc.) and the index (X register or Y register) to be used. The next 2 bytes give the base address. The following sample program shows how to print the letter A in a column entered from the keyboard. The A will be printed in row 18 (\$12).



| <i>Assembly<br/>Language</i> | <i>Machine<br/>Language</i> | <i>Comments</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JSR FC58                     | 20 58 FC                    | This new subroutine clears the text screen. It does the work of the BASIC command HOME.                                                                                                                                                                                                                                                                                                                                                                           |
| JSR FD35                     | 20 35 FD                    | Read the keyboard for the ASCII code of the column that the "A" is to be printed in. Only one digit \$0 to \$F can be entered. (Note: To enter a HEX digit from A to F, strike a key whose ASCII code's second digit is the digit from A to F that you want to enter. For example, the keys J, K, L, M, N, and O could be used to enter the HEX digits A, B, C, D, E, and F respectively.)                                                                        |
| AND #0F                      | 29 0F                       | Eliminate the left nibble of the ASCII code in the accumulator. This converts the ASCII code into the actual column number.                                                                                                                                                                                                                                                                                                                                       |
| TAX                          | AA                          | The X register is to be the index of the STA command, so the column number is transferred to the X register.                                                                                                                                                                                                                                                                                                                                                      |
| LDA #C1                      | A9 C1                       | Load the ASCII code for "A" into the accumulator.                                                                                                                                                                                                                                                                                                                                                                                                                 |
| STA 0550,X<br>RTS            | 9D 50 05<br>60              | STA is used in the absolute indexed addressing mode. The ASCII code C1 will be stored in location 0550 + Value stored in X register. For example, if the key "8" was hit, 8 would be stored in the X register and the ASCII code C1 would be stored in location 0550 + 8 = 0558. We know 0550 corresponds to the 18th row, so the STA command would cause an "A" to appear in column \$08 (the ninth column) of row 18 (the 19th row from the top of the screen). |

There are two weaknesses in the sample program given here. First, we cannot print A in columns \$10 to \$27 since we do not know as yet how to enter two-digit HEX numbers from the keyboard. In Chapter 6, we overcome this weakness. Second, we cannot enter the row number for the letter A from the keyboard. We are stuck with row 18 if we use the foregoing program. When we study the preindexed indirect addressing mode and the postindexed indirect addressing mode in Chapter 8, we will be able essentially to use STA with variable base addresses as well as a variable index. This will overcome the second weakness in the program.

## Review Questions

1. To store color number 8, you store \$\_\_\_\_\_ in location \$30.
2. The assembly language command \_\_\_\_\_ is analogous to the BASIC command GOSUB.
3. Zero-page addressing mode commands are \_\_\_\_\_ bytes long.
4. Without using a table, find the NORMAL ASCII code for the letter H.
5. The Read the Keyboard subroutine FD35 most closely resembles which BASIC command?  
(a) INPUT      (b) GET
6. What value will be stored in the accumulator as a result of these two instructions?

```
LDA #38
AND #2E
```

7. The ASCII code for F is C6. Without using a table, find the ASCII code for a FLASHing F.
8. To change a NORMAL ASCII code to an INVERSE ASCII code, you can use the assembly language instruction AND #\_\_\_\_\_.
9. Use the text screen map on page 229 to find the memory location that controls row S11, column S18, of the text screen.
10. The commands

```
LDX #05
STA 350,X
```

would store the accumulator's contents in location \$\_\_\_\_\_.

## TRUE or FALSE

11. There are one hundred zero-page locations.
12. The commands

```
LDA #05
STA 300
```

are equivalent to the BASIC command

```
POKE 768,5
```

13. If you wish to use data stored in the accumulator after calling an intrinsic subroutine, then the data should be stored in memory before calling the subroutine and reloaded into the accumulator after calling the subroutine. Some intrinsic subroutines scramble the contents of the accumulator.
14. The subroutines FDDA and FDED automatically skip to a new line each time they are used.



15. If the subroutine located at \$FD35 is used, then \$07 is stored in the accumulator when the key labeled "7" is struck.
16. The command TAY can be used in the immediate addressing mode.
17. All absolute indexed addressing mode commands are 3 bytes long.
18. Either the X register or the Y register can be used as indexes in the absolute indexed addressing mode with the commands LDA and STA.

### Programming Problems

1. Write a machine language program that draws a tic-tac-toe board.
2. Write a machine language program that generates and prints a random number between \$00 and \$0F each time it is run. (*Hint:* Use the AND command.)
3. Write a machine language program that outputs the base address corresponding to row R (column 0) of the text screen. Store the row number R in location \$390 using the monitor before you run your program.  
[*Hints:* You may want to store the most significant byte (the first two digits) of the base address corresponding to each of the 24 rows in a list stored beginning in location \$350. A list containing the least significant byte (last two digits) of each row's address might be stored beginning in location \$370. You can load a register with the row number R (use the absolute addressing mode) from location \$390. Use absolute indexed addressing to read the MSB of the base address from the first list. Print the MSB of the address after it is read. Similarly you can read and print the LSB of the desired base address from the second list.]
4. Store the color codes 00, 11, 22, . . . 99 in locations 390, 391, 392, . . . 399. Write a program that will read the keyboard for a number N between 0 and 9 (inclusive). Then plot a point in row \$14, column \$10, in the color whose code is stored in location \$390 + N.
5. Write a program that will print the character whose ASCII code is stored in location \$390 in NORMAL, INVERSE, and FLASH modes (use the monitor to store an ASCII code greater than or equal to \$C0 in location 390).

## **Chapter 4**

# **Machine Language Loops**

The real power of any computer lies in its ability to do repetitive calculations in loops and to make decisions in IF statements. In this chapter, the machine language equivalent of the loop and IF statement will be introduced. Most really interesting applications require nested loops. Although the applications in this chapter may still seem somewhat contrived, they will lay the foundations for more interesting applications.

### **Doing HEX Arithmetic Using the Monitor. Negative Numbers**

To find the operands for machine language branch commands, it will be necessary to subtract HEX addresses using HEX subtraction. Many people find HEX arithmetic confusing; however, there are monitor commands that will make the work easier.

If you type two HEX numbers separated by a plus sign, as in \*05 + 02 (no space is permitted between the first number and the + sign), and hit the RETURN key, monitor will reply with the result

=07

It should be noted that we can do only "two-digit arithmetic." That is, if you try to add \$90 + \$90 using the monitor, the result will be

\* 90 + 90  
=20



when actually the result is \$120, since

$$\begin{array}{rcccl}
 \$90 & + & \$90 & = & \$120 \\
 \uparrow & & \uparrow & & \uparrow \\
 9 \text{ in} & + & 9 \text{ in} & = & 18 \text{ in} \\
 \text{decimal} & & \text{decimal} & & \text{decimal}
 \end{array}$$

Three-digit answers cannot fit into a two-digit memory location. The monitor is certainly a help in adding two numbers, as long as you remember to keep track mentally of whether the sum is larger than \$100 and "missing the one in front" or smaller than \$100 and correct as is.

The monitor can also be used to do subtraction. For example,

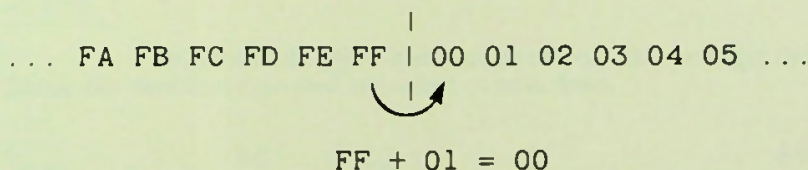
$$\begin{array}{l}
 *08 - 03 \\
 =05
 \end{array}$$

If we try to subtract a large number from a small number, we get the following result:

$$\begin{array}{l}
 *03 - 08 \\
 =FB
 \end{array}$$

This result may be interpreted in two ways. First you may think of \$FB as the result of subtracting \$103 - 08. Again we do two-digit arithmetic when we use the monitor. You can think of the computer as "borrowing one" from an imaginary byte to the left of the 03, just as you would in pencil-and-paper subtraction.

Another interpretation is possible. We can think of the possible values that a storage location can contain (\$00 to \$FF are the possible values) as "wrapping around themselves." That is, if you add 01 to FF, you come back to 00. The true result of 01 + FF is \$100, but this result is 00 in two-digit arithmetic since only the last two digits of the answer fit into storage. Hence we have the wrapping-around effect. Schematically we have



By using this diagram, you can verify that if we start at 03 and take 08 jumps to the left ("subtract 08"), we do end up with \$FB.

Notice that there are no negative numbers complete with minus signs in machine language. That is,

$$*03 - 08$$

does not produce

$$= -05$$

It is tempting to *interpret* SFB as being "equal to -5," and this is done in *certain* cases. After all, in algebra the *definition* of -5 is "that number which must be added to 5 in order to produce zero." Certainly SFB is the number that must be added to 5 in order to give 00 (in two-digit arithmetic), so FB must be "equal to -5." This interpretation will be important to us when we study branch commands later in the chapter. Thus, even though there are no numbers with minus signs in machine language, branch commands will interpret certain operands as calling for *negative*, or backward, jumps. The numbers between \$80 and \$FF inclusive (all those HEX numbers whose leftmost digit in binary is a one) are interpreted as being negative by branch commands. The rest of the numbers are considered positive. All of this will become much clearer when we discuss branch commands. For now, just bear in mind that only branching commands will interpret the numbers \$80 to \$FF as being negative and the numbers \$00 to \$7F as being positive. No other machine language commands classify numbers as either negative or positive.

We can use the first interpretation of subtraction to help us subtract four-digit HEX addresses. The procedure will be illustrated in the following three examples.

### Example 1

Subtract 0352 from 035A.

First write the problem vertically.

$$\begin{array}{r} 035A \\ -0352 \\ \hline \end{array}$$

Next break the problem into two two-digit subtraction problems.

$$\begin{array}{r|l} & \\ 03 & 5A \\ -03 & 52 \\ \hline & \end{array}$$

Working from right to left, do the subtraction using the monitor.

$$\begin{array}{r|l} & \\ 03 & 5A \\ -03 & 52 \\ \hline & 08 \end{array} \quad \text{since } 5A-52=08$$

$$\begin{array}{r|l} 03 & 5A \\ -03 & 52 \\ \hline 00 & 08 \end{array} \quad \text{since } 03-03=00$$

Thus

$$\$035A - \$0352 = \$08$$

### Example 2

Subtract 0319 from 03A7.

Write the problem vertically.



$$\begin{array}{r} 03A7 \\ -0319 \\ \hline \end{array}$$

Break the problem into two two-digit subtractions.

$$\begin{array}{r|l} & \\ 03 & A7 \\ -03 & 19 \\ \hline & \end{array}$$

Do each two-digit subtraction problem using the monitor.

$$\begin{array}{r|l} & \\ 03 & A7 \\ -03 & 19 \\ \hline & 8E \end{array}$$

↖ since  $A7-19=8E$

$$\begin{array}{r|l} & \\ 03 & A7 \\ -03 & 19 \\ \hline 00 & 8E \end{array}$$

↖ since  $03-03=00$

Thus

$$\$3A7 - \$319 = \$8E$$

### Example 3

Subtract 195 from 318.

This example shows that you must take care of "borrowing" yourself.

Write the problem vertically.

$$\begin{array}{r} 0318 \\ -0195 \\ \hline \end{array}$$

Break the problem into two two-digit subtractions.

$$\begin{array}{r|l} & \\ 03 & 18 \\ -01 & 95 \\ \hline & \end{array}$$

Do each two-digit subtraction problem using the monitor.

$$\begin{array}{r|l} & \\ 03 & 18 \\ -01 & 95 \\ \hline & 83 \end{array}$$

Before we subtract the two-digit numbers on the right, we "borrow" from the 03, since you cannot subtract \$95 from \$18. Type

$$18 - 95$$

on the monitor to perform the subtraction. The result, \$83, is the difference between \$118 and \$95.

$$\begin{array}{r} | \\ 03 | 18 \\ -01 | 95 \\ \hline | \\ 01 | 83 \end{array}$$

Complete the problem by subtracting 01 from 02. Thus

$$\$0318 - \$0195 = \$0183$$

We close this section by noting that the monitor does not do multiplication (nor division). Thus

$$*2*2$$

is illegal. Also, the monitor does *not* do "chain" calculations. If you type

$$*1+2+3$$

the result is

$$=04$$

since any attempt to add more than two numbers will result in only the first and last numbers being added.

### Saving Programs in Binary

We can use our subtraction skills to help us save machine language programs on our disks. The command used to save programs is BSAVE (B stands for "binary"). It works just as SAVE does, except that there are two extra parameters. The syntax of BSAVE is

BSAVE Name, A\$ Starting Address, L\$ Program Length



Here Name is the name you give your program. A stands for address. The \$ indicates that the address to be given will be in HEX (you could give the address in decimal, if you left off the \$). The address A you must supply the BSAVE command with is the address of the first byte in your program. L stands for "length" of the program in bytes and the \$ again indicates that the length is to be given in HEX (you could give the length in decimal, if you left off the \$). To find the length of short programs, you can actually count the number of bytes. For longer programs, use

$$\text{Length} = \text{Address of last byte} - \text{Address of first byte} + 1$$

### Example

Save a program named KOUNT that starts in location \$1020 and whose last byte is in location \$1109.

$$\text{Length} = \$1109 - 1020 + 1 = \text{E9} + 1 = \text{EA}$$

Use the monitor  
to subtract  
1020 from 1109



so the correct command is

```
BSAVE KOUNT, A$1020, L$EA
```

The BSAVE command can be used from either Applesoft or the monitor program. More detailed information on BSAVE can be found in the *DOS Manual* (page 92).

To load the program into memory, simply type

```
BLOAD KOUNT
```

No parameters are needed. The program will be loaded into the same memory locations it was BSAVED from. The command

```
BRUN KOUNT
```

could be used both to load the program into the same memory locations it was BSAVED from and to run the program.

If BSAVE is used while you have a program in memory, then the program will still be in memory after BSAVE is executed. Sometimes DOS will not be booted up when you go to BSAVE your program (this will be the case, for example, if you forgot to boot DOS before you entered your program). If PR#6 is used to boot DOS while a machine language program is in locations \$300 to \$3FF, then the program will be lost. If you must boot DOS after entering a machine language program, move the program to location \$1000 before booting DOS, as this location is not affected when DOS is booted. You can move your program back to location \$300 after you boot DOS.

### Machine Language Loops

In machine language, there is no equivalent for FOR-NEXT loops. Machine language loops are closely related to the IF-GOTO type of loops that you learned about before you were taught the FOR-NEXT combination. For example, the machine language version of the program

```
10 FOR I=1 TO 10
20 PRINT "*";
30 NEXT I
```

will closely resemble

```
10 I=0
20 PRINT "*";
30 I=I+1
40 IF I<>10 THEN 20
```

Unfortunately there is no exact machine language equivalent for the IF statement either. Two machine language commands, a compare command and a branch command, together will take the place of a BASIC IF statement.

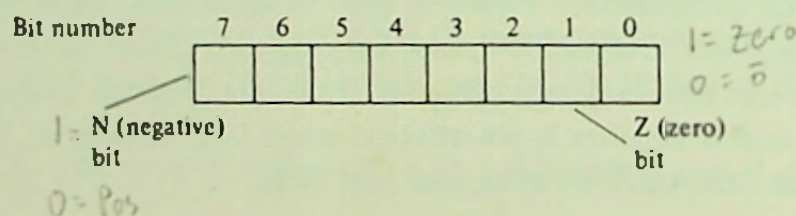
IF I <> THEN 20

A machine language compare command takes the place of the first part of the IF statement.

The machine language branch command takes the place of the last part of the IF statement. It branches depending on the results of the compare command.

### The Compare Command and the Processor Status Register

In Apple's memory is a special register called the processor status register (P register). There are 8 bits of information in this memory location, as in all of Apple's memory locations. The processor status register can be visualized as follows





Two of the bits of the P register are labeled here. The other bits also have labels, but we do not discuss them in this chapter. A complete schematic diagram of the processor status register is given at the beginning of Appendix B.

Each time an instruction is executed, the result (if an arithmetic operation such as addition or subtraction is performed) or the number moved (if a command such as LDA or LDX is used) is examined. If the result is negative (between \$80 and \$FF inclusive), then the N bit is set equal to 1. If the result is zero, then the Z bit is set equal to 1. If both the N bit and Z bits are set equal to zero, then the result is strictly positive (between \$01 and \$7F inclusive). The branch commands to be described shortly use the Z and N bits to decide whether a result is positive, zero, or negative. Programmers usually do not deal with the processor status register directly. It is described here merely to give you a better idea of what is going on.

One very special command that produces positive, negative, or zero results is the compare command. There are three compare commands—CMP, CPY, and CPX, for comparing the contents of the accumulator, Y register and X register, respectively. How the CMP command (CoMPare the contents of the accumulator) works is described in the following; the other two compare commands function in the same way. Three examples illustrate the use of CMP.

In the immediate addressing mode, the command

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                             |
|--------------------------|-------------------------|-------------------------------------------------------------|
| CMP #18                  | C9 18                   | The op code for CMP in the immediate addressing mode is C9. |

tells Apple to decide whether the expression

Contents of accumulator minus \$18

is positive, negative, or zero. The appropriate bits in the P register are then changed. A branch command always follows a CMP command, and the branch command will look at the P register to decide whether or not a branch (conditional jump) will take place, as described shortly. It should be emphasized here that the CMP command in no way affects the actual contents of the accumulator. \$18 is not actually subtracted from the accumulator by the CMP command. When the expression

Contents of accumulator minus \$18

is evaluated, the subtraction can be thought of as taking place in a separate location with the results being used to set the appropriate bits in the P register. The accumulator's contents are left unmolested. For example, if the accumulator held \$01 before the command CMP #18 was executed, it would still hold \$01 after the CMP command was executed. Since the value of  $\$01 - \$18$  is negative, the CMP command would set the N bit of the P register equal to one in this example.

The CMP command can be used in the absolute addressing mode.

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                            |
|--------------------------|-------------------------|------------------------------------------------------------|
| CMP \$350                | CD 50 03                | CD is the op code for CMP in the absolute addressing mode. |

The CMP command here will look at the value of the expression

Contents of the accumulator – Contents of memory location \$350

and change the appropriate bits in the P register, depending on whether the expression is positive, negative, or zero. The contents of both the accumulator and memory location \$350 are left untouched by the command CMP \$350.

The CMP command can also be used in absolute indexed addressing mode, with either the X or Y register serving as the index. For example, the commands

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                                    |
|--------------------------|-------------------------|--------------------------------------------------------------------|
| LDX #05                  | A2 05                   | DD is the op code for CMP in the absolute indexed addressing mode. |
| CMP 800, X               | DD 00 08                |                                                                    |

would compute the value of the expression

Contents of the accumulator – Contents of location \$805  
(since  $800 + X = 800 + 5 = 805$ )

and set the appropriate bits in the P register, depending on whether the expression was positive, negative, or zero.

The commands CPY and CPX work analogously to the CMP command, except that the CPX and CPY (ComPare X register and ComPare Y register) can be used only in the immediate addressing mode, absolute addressing mode, or zero-page addressing mode. For example, in

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                             |
|--------------------------|-------------------------|-------------------------------------------------------------|
| CPX #05                  | E0 05                   | E0 is the op code for CPX in the immediate addressing mode. |

the command CPX #05 would compute the value of the expression

X register – 05

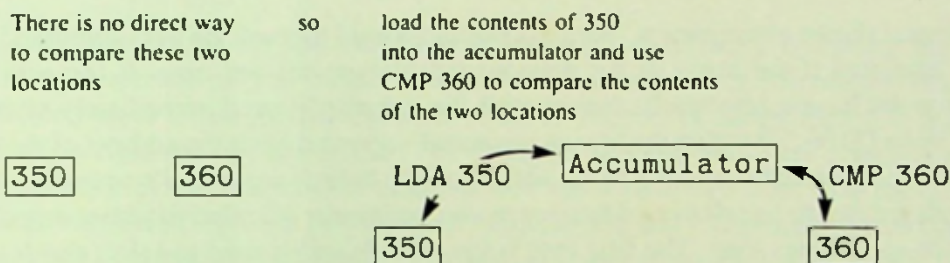
and set the appropriate bits in the P register.



It should be noted that there is no direct way to compare the contents of two memory locations. The accumulator (or the X register or the Y register) must be used as a go-between. For example, the commands

| <i>Assembly Language</i> | <i>Machine Language</i> |
|--------------------------|-------------------------|
| LDA 350                  | AD 50 03                |
| CMP 360                  | CD 60 03                |

could be used to compare location 350 with location 360 using the accumulator as a go-between. Schematically we have



## Branch Commands and the Relative Addressing Mode

The branch commands discussed here are

- BNE Branch if the result of the previous calculation is *Not Equal* to zero.
- BEQ Branch if the result of the previous calculation is *EQual* to zero.
- BPL Branch if the result of the previous calculation is *PLus* (greater than or equal to zero).
- BMI Branch if the result of the previous calculation is *MInus* (negative).

The op codes of these commands can be found in Appendix B.

Compare commands are always followed by branch commands, although branch commands are not always preceded by compare commands. Essentially a compare command stores signals specifying "positive," "negative," or "zero" in the P register. A branch command makes a conditional jump to a section of the program determined by the command's operand. The decision to jump or not to jump is made by using the signals stored in the P register. For example, if the Z bit in the P register is zero, then the command BNE will cause a branch to occur, since the result of the previous calculation was not zero. If the Z bit had been one instead, then no branch (jump) would have occurred. Control would have been passed to the command following the BNE command.

Before showing the interpretations attached to various compare command/branch command combinations, you must know how the operand of a branch command works. The operand of a command such as BNE tells the computer how far to jump (forward or

*z = 0 means result ≠ 0  
∴ BNE - True*

backward) to reach the next command that needs to be executed (when the result of the previous calculation is not zero). For example,

| <i>Assembly Language</i> | <i>Machine Language</i> |
|--------------------------|-------------------------|
| BNE \$352                | D0 06                   |

The assembly listing of a branch command is a little unusual. A complete explanation will be given shortly.

the command shown above means "jump six places forward to reach the next command that is to be executed if the result of the previous calculation was not zero. If the previous calculation did have a zero result, then execute the command stored immediately after the operand 06 in D0 06." Because the branch command's operand gives the address of the next command to be executed *relative* to the address of the branch command's operand, these commands are said to be *relative addressing mode* commands. All relative addressing mode commands are 2 bytes long. The first byte is the op code and is used to select the desired branch condition (BNE, BPL, etc.). The second byte is an operand that tells Apple the size and direction of the conditional jump to be made. The designers of machine language *could* have designed BNE to be an absolute addressing mode command. Then D0 52 03 (this is *not* a valid command) would have meant "branch to location \$352 if the previous result was not zero." The command might have been easier to understand, since a definite destination address is given. The hypothetical absolute addressing mode branch command is analogous to IF . . . THEN 100; the number 100 tells you the line number to execute next. For reasons described later, branch commands do *not* work in this way. Rather than supplying BNE with the *address* of the branch destination, the Apple microprocessor expects you to supply *directions* to get to the branch destination. These directions are of the form "take five steps forward from your present location" or "take 17 steps backward from your present location." Study the following list to understand how to count forward and backward steps or jumps in branch commands. You must be able to do this if you are to figure the operand of your branch command.

34C .

34D This location is reached by taking five backward steps.

34E This location is reached by taking four backward steps.

34F This location is reached by taking three backward steps.

350 D0 (op code for BNE stored here); this location is considered to be two backward steps from the end of the branch command.

351 Operand for BNE stored here; this location is considered to be one backward step from the end of the branch command.



- 352 This location is considered to be zero forward steps from the end of the branch command.
- 353 This location is considered to be one forward step from the end of the branch command.
- 354 Take two forward steps to reach this spot.
- 355 Take three forward steps to reach this spot.
- 356 Take four forward steps to reach this spot.
- 357 .

It is important to realize that the memory location immediately following the branch command's operand is "zero steps" from the branch command. If you need to branch to a place fairly close to this location, you can count the number of jumps needed to reach your destination right on Apple's screen. For example, suppose you were typing a machine language program into the monitor as shown here and wish to figure out what the operand of the BNE command (op code D0) stored in locations 352 and 353 should be (the operand is represented by question marks here). You wish to branch to the circled location if the result of the LDA \$05 command is not zero.

```
350:A5 05 D0 ?? AD 92 03 20
358:DA FD . . .
```

To find the operand, simply count the jumps required to get to location 357. Use the method suggested here:

```
350:A5 05 D0 ?? AD 92 03 20
           ↑   ↑   ↑   ↑
           zero one two three
           jumps jump jump jumps
```

Thus the ?? in location 353 should be replaced with the HEX number 03.

Calculating backward jumps is just as easy. Suppose you wish to calculate BNE's operand in the situation shown in the following. You wish to branch from the BNE command stored in locations 357 and 358 to the circled byte if the result of the previous calculation is not a zero.

```
350:A6 05 98 20 DA FD CA D0
358:?? AD 92 03
```

Count the jumps as before. The location corresponding to zero jumps is location 359 (which holds the byte AD).

|       |      |       |                |              |               |               |                |              |
|-------|------|-------|----------------|--------------|---------------|---------------|----------------|--------------|
|       |      |       | seven<br>jumps | six<br>jumps | five<br>jumps | four<br>jumps | three<br>jumps | two<br>jumps |
|       |      |       | ↓              | ↓            | ↓             | ↓             | ↓              | ↓            |
| 350 : | A6   | 05    | 98             | 20           | DA            | FD            | CA             | D0           |
| 358 : | ??   | AD    | 92             | 03           |               |               |                |              |
|       | ↑    | ↑     |                |              |               |               |                |              |
|       | one  | zero  |                |              |               |               |                |              |
|       | jump | jumps |                |              |               |               |                |              |

We need an operand corresponding to seven backward jumps in this case. The machine language equivalent of -7 will be the correct operand. You can use the monitor command

```
*0-7
=F9
```

to deduce that the question marks appearing in location 358 should be replaced by F9. It should be pointed out that it is fine to use commands such as

```
*0-7
```

while you are entering a machine language program using the monitor. If you are in the middle of entering a line of 8 bytes, simply hit the RETURN key, use the monitor subtraction command, and then finish the line of 8 bytes. For example, the sequence

```
*350:A6 05 98 20 DA FD CA D0
*0-7
=F9
```

```
*358:F9 AD 92 03 8D 95 03 AD
```

is a perfectly valid way to enter bytes in locations 350 to 35F.

In programs with longer branches, counting bytes is far too tedious. You can use the following rules to find the operand of a branch command.

For forward jumps:

Operand = Address of destination - Address of byte immediately following the branch command's operand

For backward jumps:

Number of backward steps = Address of byte following branch command's operand - Address of destination



Once the number of backward steps is known, use the monitor's subtraction capabilities to find the operand, by typing

\*0 - number of backward steps

When doing monitor subtraction, you should always type

\*0 - number of backward steps

instead of just

\* - number of backward steps

to ensure correct results all the time. That is, you would type

\*0 - 7

not

\* -7

to find the operand corresponding to a jump of seven backward steps.

### Example

The first byte of a BNE command is stored in location 362. The destination of the BNE command is 385. To find BNE's operand, we subtract 364 from 385, since the byte following BNE's operand is stored in location 364.

$$\begin{array}{r} 3 \ 85 \\ -3 \ 64 \\ \hline 21 \text{ (since } 85 - 64 = \$21) \end{array}$$

Thus the operand 21 should be stored in location 363.

### Example

The first byte of a BEQ command is stored in location 354. The destination of the BEQ command is location 328. To find the operand, we subtract \$328 from \$356, since the address immediately following the BEQ command's operand is \$356.

$$\begin{array}{r} 3 \ 56 \\ -3 \ 28 \\ \hline 2E \end{array}$$

Therefore the number of backward steps needed to reach the destination is \$2E. To find the actual operand corresponding to \$2E backward jumps, use monitor's subtraction capabilities. If you type

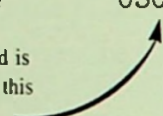
```
*0-2E
```

the result is

```
=D2
```

Thus the operand to be stored in location \$355 is \$D2. Some readers may be puzzled by this question: How will we know the destination address for forward branches? When the monitor gives you an assembly language listing, the address of each command's op code (the first byte) is displayed, as in

|               |       |          |            |
|---------------|-------|----------|------------|
| Address of    | 0300- | AD 50 03 | LDA \$0350 |
| first byte    | 0303- | 8D 70 03 | STA \$0370 |
| in each       |       |          |            |
| command is    |       |          |            |
| given in this |       |          |            |
| column        |       |          |            |



If you are like most people, when you begin to write your program you write down something like

```
LDA 350
STA 370
```

and you do not bother to figure out the address of the op code in each command. After all, the only address you need to know is the address of the very first command in your program, which will usually be \$300. As you type commands 1 byte at a time in consecutive memory locations, addresses for each command will take care of themselves. If you run into a BNE command, however, you need to know the address of the destination of the branch. If the branch is a backward branch, there will be no problem, since the destination command's op code and operands have already been typed in and are there for you to see. For forward jumps, this is not the case, however. Since the destination command will not have been typed in as yet, you will not know the address of its first byte. One way around this problem is to fill in the branch command's operand with any number (as a placeholder). After the rest of your program has been placed in memory and the starting address of each command can be seen, you can always go back and correctly find the operand for the forward jump. The operand you used as a placeholder then easily can be replaced with the correct operand by using the monitor.

One feature of the monitor that is a big help in debugging is the method used to list branch commands. If you enter the machine language codes

```
300:D0 05
```



these bytes will not be listed as

```
300- D0 05      BNE 05
```

Instead the listing will look like this:

```
0300- D0 05      BNE $0307
                        ↗
                    destination address
```

You can see that the listing gives the destination address of BNE, which is 0307 (=0302 + 5) in this case. In fact, when the mini-assembler (discussed in the next chapter) is used, you will actually type

```
BNE 0307
```

to enter the BNE command and the mini-assembler will calculate the operand for you. Note that even with the mini-assembler, forward jumps still will be troublesome since the destination address for the forward jump will not be known at the time the branch command is typed in. The strategy again will be to type in a placeholder as a destination address. You can calculate the correct operand after the rest of the program is entered into memory.

You might have noticed that the maximum number of forward jumps a branch command can make is limited to \$7F (127) because, if an operand from \$80 to \$FF is used, the branch command will consider the operand negative and jump backward, not forward. (Similar restrictions exist on backward jumps. The maximum number of backward jumps a branch command can make is limited to \$80.) Normally this will not cause a problem, as you rarely will need to jump forward more than \$7F times. If a need does arise to jump forward more than \$7F times, you can use two consecutive branch commands. For example, to jump forward \$90 times, you might branch forward \$70 times first. In the destination address of the first branch command, you can store another branch command that branches forward \$20 more times. The method just described works because the first branch command has no effect at all on the P register. Therefore, when the second branch command is reached, it will branch just as the first command did. In general, two branch commands can be used in tandem, sharing the same compare command. This fact will prove useful shortly.

## Using Compare and Branch Statements Together

Our goal in this section is to pull together all the facts we know about compare and branch statements so as to mimic such BASIC statements as

```
IF X > Y THEN GO TO . . .
IF X < =Y THEN GO TO . . .
```

The following discussion will show the various combinations of statements that are possible and their effects. In the discussion, the shorthand notation

IF \$300 < \$302 THEN GO TO \$355

will be used to mean, "If the contents of location \$300 are less than the contents of location \$302, then jump to the command stored in location \$355." Of course, HEX addresses cannot be used in BASIC statements, but the notation used here will show what the machine language commands do in terms that are familiar to the reader. It should be understood that program segments are being shown schematically here. Complete program examples will follow. Arrows are used in the examples to clarify the flow in the program; they would not appear in a real assembly language listing. Numerical labels that do appear in a normal listing for every command are not shown here. Such labels are used on key commands only.

### Example 1

| Assembly Listing                         | Comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 300- LDX #00                             | X is initialized to 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 302- .<br>. body of the<br>. loop<br>INX | The new command INX (op code E8) increments the value stored in the X register by one. It is equivalent to the BASIC statement $X = X + 1$ . The INX command is an implied addressing mode command, since no operands need be given. The address of the affected register is <i>implied</i> in the mnemonic code INX. Notice that X is being used as a counter that counts the number of times the body of the loop has been executed. For example, after the body of the loop has been executed once, the value stored in X is one. After the body of the loop has been executed twice, the value stored in X is two, and so on. |
| CPX #05                                  | This command calculates the value of<br><br>Contents of X register - \$05                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| BNE \$302<br>RTS                         | This result will not be zero when $X = 1, 2, 3, \text{ or } 4$ . When $X = 5$ , the value of the expression will be 0. Therefore the branch statement will cause a branch to the top of the loop until the body of the loop has been executed \$05 times.                                                                                                                                                                                                                                                                                                                                                                         |

Thus the two commands



```
CPX #05
BNE $302
```

in assembly language closely resemble the "BASIC statement"

```
IF X-05 <> 0 THEN GO TO $302
```

or

```
IF X <> 05 THEN GO TO $302
```

You should also know that there is a command analogous to INX that increments the Y register, INY (op code C8). There is also a command INC that increments a regular memory location by one. The INC command can be used in absolute addressing mode or absolute indexed addressing mode. There is no command "INA" to increment the accumulator, however.

### Example 2

Example 1 also could have been written as follows:

| Assembly Language | Comments                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 300- LDX#00       |                                                                                                                                                                                                                                                                                                                                                                                                     |
| 302- .            |                                                                                                                                                                                                                                                                                                                                                                                                     |
| . body of the     |                                                                                                                                                                                                                                                                                                                                                                                                     |
| . loop            |                                                                                                                                                                                                                                                                                                                                                                                                     |
| INX               |                                                                                                                                                                                                                                                                                                                                                                                                     |
| CPX #05           | The command CPX#05 calculates the value of the expression                                                                                                                                                                                                                                                                                                                                           |
|                   | Contents of X register - 05                                                                                                                                                                                                                                                                                                                                                                         |
| ← BMI \$302       | When the X register contains 1, 2, 3, or 4, the value of this expression is negative and the BMI command will branch back to the top of the loop. When X finally equals 05, then the value of the expression is zero, so that the branch command will <i>not</i> branch back to the top of the loop. Thus the body of the loop is executed five times by this program and the program of Example 1. |
| RTS               |                                                                                                                                                                                                                                                                                                                                                                                                     |

The assembly language commands

```
CPX #05
BMI $302
```

resemble the "BASIC statement"

```
IF X - 05 < 0 THEN GO TO $302
```

or

```
IF X <05 THEN GO TO $302
```

It should be further emphasized that the X register counts the number of times the body of the loop has been executed. When X is used in this way, it simplifies the task of understanding the program. If the first command had read LDX#01 instead of LDX#00, then the value of X would have equaled the number of times the body of the loop was executed plus one. Initializing X to zero at the beginning of the program and using it as a counter as illustrated in Examples 1 and 2 makes understanding the program logic easier and reduces the risk of programming errors.

### Example 3

There is another slightly more compact way to write a loop that executes a group of statements five times.

| Assembly Language | Comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 300- LDX #05      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 302- .            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| . body of the     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| . loop            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| DEX               | The new command DEX (op code CA) decreases the contents of the X register by one. There is an analogous command DEY (op code 88) that decreases the contents of the Y register by one. In addition, the assembly language command DEC can be used to decrease by one any of Apple's numbered memory locations. DEC can be used in the absolute addressing mode or the absolute indexed addressing mode.                                                                                                                                                                                                                                                                                                                                                                                                        |
| BNE \$302         | Notice that there is no compare command preceding the BNE command. Recall that BNE means "branch if the result of the previous calculation is not zero." In this case, the previous calculation is a DEX command rather than a compare command. After five times through the loop, the DEX command will have subtracted a total of five from the original contents of the X register. Therefore, after executing the body of the loop five times, the value stored in the X register will be zero. At this point, the BNE command will not cause a branch to the top of the loop and execution will be terminated. Although loops employing DEX rather than INX may be a bit more difficult for the beginning programmer to understand, they do have the advantage of being shorter since CPX does not appear. |
| RTS               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |



In this example, the BNE \$302 command is conceptually equivalent to the "BASIC statement"

```
IF X <> 0 THEN $302
```

#### Example 4. Behavior of BPL

The BPL command "branches if the result of the previous calculation is plus." Unfortunately the BPL command bases its decision on whether or not to branch by referring to the N bit in the P register. This means that even when the result of the previous calculation is zero, BPL will execute a branch, since 0 is not negative. Perhaps a better name for BPL would have been BNN: "Branch if the result of the previous calculation was nonnegative." Suppose we use BPL to replace BNE in the program of Example 3.

| Assembly Language | Comments                                                                                                                                                                                                                                                                                                                                                                                           |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 300- LDX #05      |                                                                                                                                                                                                                                                                                                                                                                                                    |
| 302- .            |                                                                                                                                                                                                                                                                                                                                                                                                    |
| . body of the     |                                                                                                                                                                                                                                                                                                                                                                                                    |
| . loop            |                                                                                                                                                                                                                                                                                                                                                                                                    |
| DEX               |                                                                                                                                                                                                                                                                                                                                                                                                    |
| BPL \$302         | After executing the body of the loop five times, the contents of the X register will be zero at the bottom of the loop. Since zero is a "plus" number, the command BPL will branch back up to the top of the loop. After executing the loop for a sixth time, the value stored in the X register will be \$FF, which is not considered "plus" by BPL. At this point, execution will be terminated. |
| RTS               |                                                                                                                                                                                                                                                                                                                                                                                                    |

In this example, the BPL \$302 command is conceptually equivalent to

```
IF X > = 0 THEN $302
```

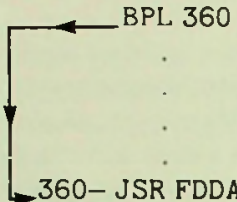
Branch commands can be used in places other than machine language loops. We can use compare and branch commands to do the work of such statements as

```
IF M < N THEN GO TO 80
```

in BASIC. The following examples illustrate the general method.

#### Example 5

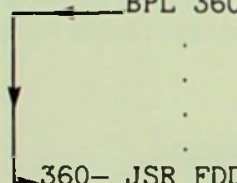
```
IF $390 > = $391 THEN $360
```

| Assembly Language                                                                                                                                            | Comments                                                                                                                                                                                                                                                                                                                 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 340- LDA 390<br>CMP 391                                                                                                                                      | CMP will calculate the value of the expression<br><br>Value stored in 390 - Value stored in 391                                                                                                                                                                                                                          |
|  <pre>       BPL 360       .       .       . 360- JSR FDDA           </pre> | <p>If the value of this expression is greater than or equal to zero, the branch command will "GOTO" address 360. Thus, CMP and BPL's actions can be thought of as being analogous to</p> <p>IF 390 - 391 &gt; = 0 THEN \$360</p> <p>or</p> <p>IF 390 &gt; = 391 THEN 360</p> <p>or</p> <p>IF 391 &lt; = 390 THEN 360</p> |

Also note that the program

```

340- LDA 391
      CMP 390
      BPL 360
      .
      .
      .
360- JSR FDDA
          
```



would not be equivalent to the first program. The order in which you load the accumulator and compare is crucial. In the second program, CMP and BPL's actions are analogous to

IF \$391 - \$390 > = 0 THEN \$360

or

IF \$391 > = \$390 THEN \$360

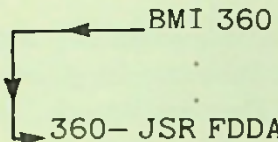
or

IF \$390 < = \$391 THEN \$360



## Example 6

IF \$390 < \$391 THEN \$360

| Assembly Listing                                                                                                                                             | Comments                                                                                                           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| 340- LDA 390<br>CMP 391                                                                                                                                      | The compare command will compute the value of the expression<br><br>Contents of location \$390 - Contents of \$391 |
|  <pre> graph TD     BMI[BMI 360] --&gt; JSR[360- JSR FDDA]           </pre> | If the value of this expression is negative, the branch command will transfer control to location 360.             |

Therefore, the BMI-CMP combination is conceptually equivalent to

IF \$390-\$391 < 0 THEN GO TO \$360

or

IF \$390 < \$391 THEN GO TO \$360

or

IF \$391 > \$390 THEN GO TO \$360

You can change the sense of the inequality in the program by replacing

```
LDA $390
CMP $391
```

by

```
LDA $391
CMP $390
```

With this replacement, you should see that the BMI-CMP combination does the work of

IF \$391 - \$390 < 0 THEN GO TO \$360

or

IF \$391 &lt; \$390 THEN GO TO \$360

or

IF \$390 &gt; \$391 THEN GO TO \$360

**Example 7**

This is another combination equivalent to

IF \$390 &lt; \$391 THEN \$360

You should not be confused by the fact that there seem to be many combinations of branch commands and compare commands that accomplish the same result. The examples given so far do not show the only way to replace a given IF statement. They are intended for illustration purposes only. To emphasize this point, consider

*Assembly Listing**Comments*

340- LDA 391

343- CMP 390

The expression

Value stored in \$391 - Value stored in \$390

will be used by both branch statements to determine if a branch occurs.

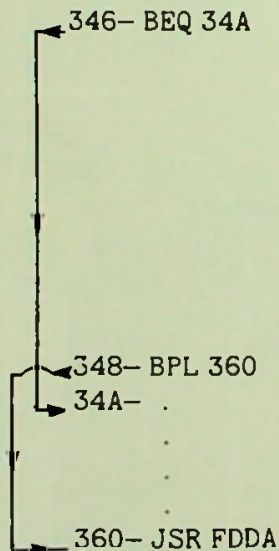
This BEQ statement will branch to \$34A when the value of the expression is zero. The CMP-BEQ combination does the work of

IF 391 - 390 = 0 THEN 34A

or

IF 391 = 390 THEN 34A

Normally the BPL command would cause a branch whenever the expression shown above was positive or zero. Since the BEQ command "siphons off" those cases in which the value of the expression is zero, we may conclude that BPL will branch to 360 only when the value of the expression is positive.





Thus, with this particular program structure, the only time a branch to location 360 occurs is when

$$\text{Value stored in 391} - \text{Value stored in 390} > 0$$

The net effect of the compare command and two branch commands is thus

IF \$391 - \$390 > 0 THEN GO TO 360

or

IF \$391 > \$390 THEN GO TO 360

or

IF \$390 < \$391 THEN GO TO 360

### Unconditional Jumps

Branch statements cause jumps to occur when certain conditions are satisfied. In certain instances, it may be desirable to jump every time a certain place in the program is reached. In BASIC, the GOTO statement is used to make unconditional jumps from one part of a program to another. The assembly language equivalent of GOTO is JMP (short for *JuMP*, op code 4C). The operand of a JMP command is a 2-byte address giving the destination of the jump. For example,

| <i>Assembly Language</i> | <i>Machine Language</i> |
|--------------------------|-------------------------|
| JMP 390                  | 4C 90 03                |

means "jump to the command stored in location \$390." The operand in JMP tells the Apple what statement to execute next, instead of how to get to this statement as the operand of branch commands did.

Whenever possible, relative addressing mode commands should be used, even though JMP seems much easier to understand. The program structure

```

350- LDX #00
352- .
      .
      .
      INX
      CPX #05
      BNE $352
      RTS

```

is far more desirable than

```

350- LDX #00
352- .
      .
      .
      INX
      CPX #05
      BEQ $380
      JMP 352
380- RTS

```

Suppose in the first program you needed to add the command LDA 390 at location \$340. All the commands presently stored following this location would have to be "pushed up three spaces" to make room for the 3-byte command LDA 390. You should see that this editing process would not affect the branch command BNE since the number of bytes between the branch command BNE and the first statement in the body of the loop is unchanged by merely moving every statement in the loop forward three memory locations. The operand formerly used in the BNE command would continue to be valid. The assembler listing might change to

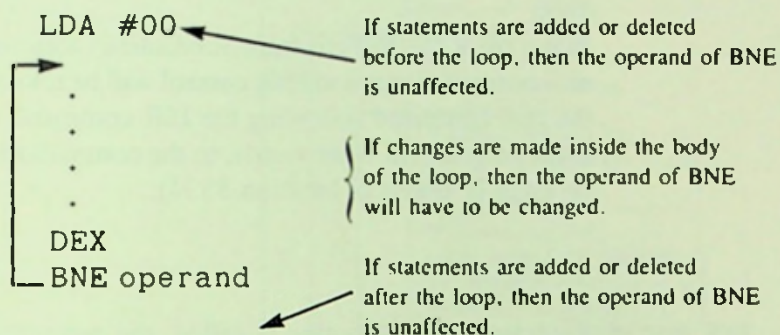
```

353- LDX #00
355- .
      .
      .
      INX
      CPX #05
      BNE 355
      RTS

```



but the BNE statement would still branch to the same statement as before. The only time the operand of a relative addressing mode command would have to be modified would be if a statement were added or deleted from memory *between* the branch command and its destination address. Schematically we have



Contrast that with the effect of editing a program containing a `JMP` command, as in the second program. After editing this program, the first statement in the body of the loop would be stored in location \$355 instead of \$352. After editing, `JMP $352` would cause a jump to the place where the first statement in the body of the loop used to be, instead of the first statement's new location. In a program with many `JMP` commands, the programmer would have to modify the operands of each `JMP` command every time the program was edited.

We summarize the discussion by saying that the use of relative addressing mode commands gives our program "portability"—that is, if the program is moved in memory, it will still function as before. This portability makes editing much easier.

## Subroutines in Machine Language

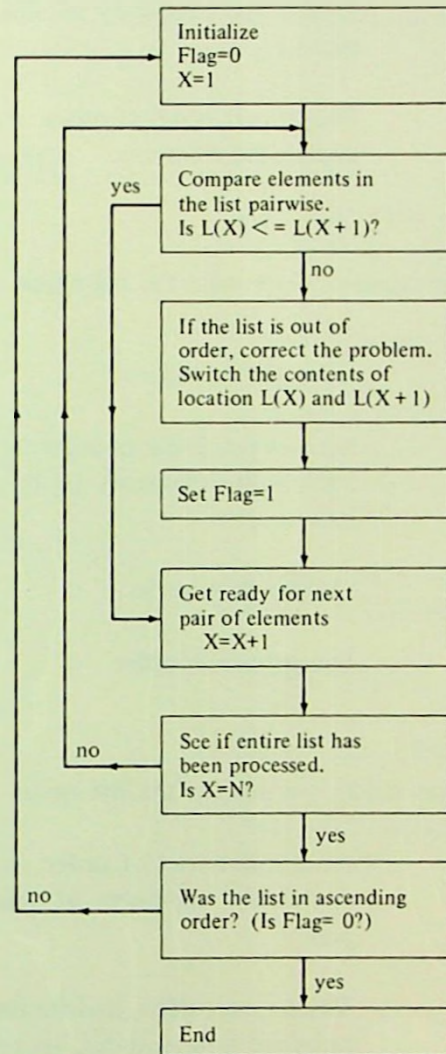
Up to now, we have been using the `JSR` (Jump to SubRoutine) command to access subroutines built into Apple's memory. You can also use the `JSR` command to write your own subroutines in a manner exactly comparable to the way you did in BASIC. The following illustrates the approach.

| Assembly Language                                                                                                                                                                                       | Comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>↑</p> <p>331- JSR 800</p> <p>334 .</p> <p>MAIN PROGRAM</p> <p>36A- JSR 800 ←</p> <p>36D .</p> <p>385- JSR 800</p> <p>388 .</p> <p>3B0- RTS</p> <p>800</p> <p>SUBROUTINE</p> <p>850- RTS</p> <p>↓</p> | <p>This command transfers control to the statements stored in the block of memory starting with location \$800.</p> <p>When the RTS (<i>ReTurn</i> from Subroutine) command is encountered in location 850, control will be returned to the first command following the JSR command in the main program (in other words, to the command whose op code is stored in location \$334).</p> <p>Each time the subroutine is called, the computer will "remember" where it was called from. The RTS command located in address \$850 will return control to the statement following the appropriate JSR command.</p> <p>The RTS command in the <i>main</i> program returns control to the monitor program after the main program has been executed.</p> <p>The last command in every subroutine is RTS.</p> |



**A Step-by-Step Programming Example: The Bubble Sort**

Suppose a list of  $N$  numbers is stored in memory and we wish to sort this list in ascending order. We can follow the algorithm outlined in the flowchart.



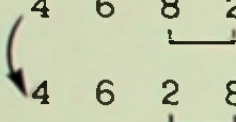
In the bubble sort, you compare all pairs of elements that are next to each other in the list. Each time you find a pair that is in the wrong order (right-hand element smaller than the left-hand element), you switch these elements and set a *flag* equal to one. After processing all pairs in the list, you look at the flag. If it is zero, then you know that no switches were made. In this case, the list must have been in the correct order. If the flag is one, then at least one pair was out of order. In this case, you go back to the beginning of the list and start the procedure over again. The list is processed until it is in order (in which case no switches will be necessary and the flag will be zero after the list has been processed). The procedure is illustrated as follows:

4 6 8 2 7  

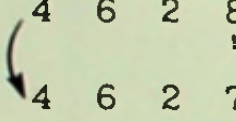

First pair in order.

4 6 8 2 7  


Second pair in order.

4 6 8 2 7  
  
 4 6 2 8 7  


Third pair is out of order, so switch the elements of this pair.

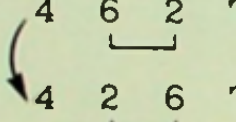
4 6 2 8 7  
  
 4 6 2 7 8  


Fourth pair is out of order, so switch the elements.

Since at least one switch was made, we process the list again.

4 6 2 7 8  


First pair in order.

4 6 2 7 8  
  
 4 2 6 7 8  


Second pair is out of order, so switch the elements of this pair.

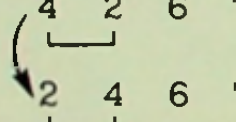
4 2 6 7 8  


Third pair in order.

4 2 6 7 8  


Fourth pair in order.

Since at least one switch was made, we process the list again.

4 2 6 7 8  
  
 2 4 6 7 8  


First pair is out of order, so switch the elements of this pair.

The second, third, and fourth pairs are now in order, so no switches will be made.

At least one switch was made, so the list will be processed again. Since the list is now in the correct order, no further switches will be made and execution will be terminated.

To write the machine language program, first jot down the assembler or mnemonic codes for each block in the flowchart. It is also a good idea to decide how you want to make use of memory. Write down the key locations and their uses to help yourself and anyone else who needs to follow your program.



## Memory Use

390—Flag.

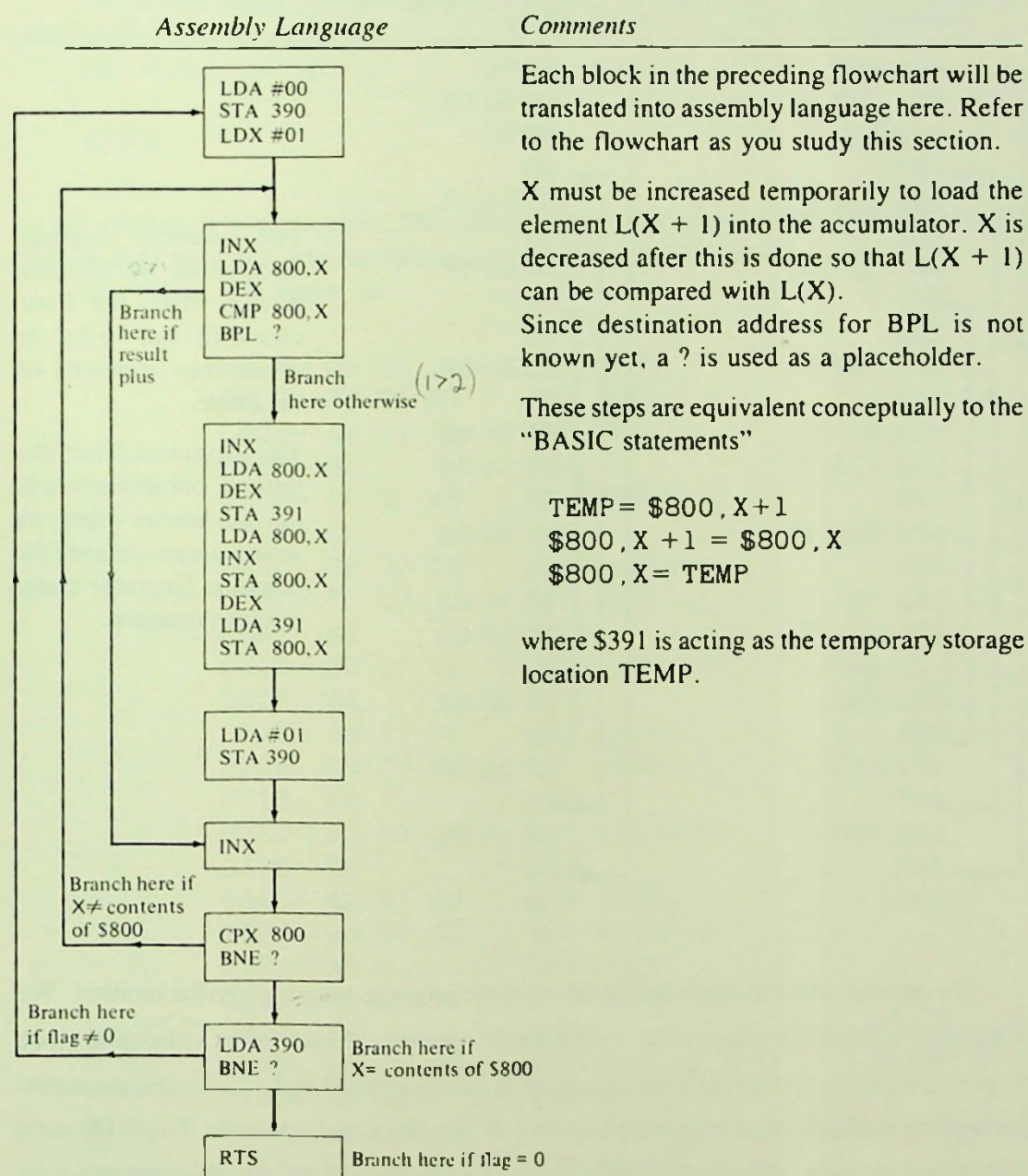
391—Temporary storage used when elements in the list need to be switched.

800—Stores the length of the list to be sorted.

801 to 801 + length of list—Storage for elements in the list L(X).

X register—"Subscript" used to index the list.

300—Starting location for the program itself.



Most of the work is now done. One of the beauties of machine language is that there is a virtual one-to-one correspondence between the flowchart and the assembly language version of a program. You should see that following the flowchart block by block makes it possible to write down the assembly language version of the program with a minimum of confusion. The next step is to look up the op codes for each of the assembly or mnemonic codes in the program (see Appendix B for the op codes).

| <i>Assembly Language</i> | <i>Machine Language Codes</i> |                                                                                                                              |
|--------------------------|-------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| LDA #00                  | A9 00                         |                                                                                                                              |
| STA 390                  | 8D 90 03                      |                                                                                                                              |
| LDX #01                  | A2 01                         |                                                                                                                              |
| INX                      | E8                            |                                                                                                                              |
| LDA 800, X               | BD 00 08                      |                                                                                                                              |
| DEX                      | CA                            |                                                                                                                              |
| CMP 800, X               | DD 00 08                      |                                                                                                                              |
| BPL ?                    | 10 ?                          | The assembly language instructions were copied directly from the flowchart in the order in which they appeared on the chart. |
| INX                      | E8                            |                                                                                                                              |
| LDA 800, X               | BD 00 08                      |                                                                                                                              |
| DEX                      | CA                            |                                                                                                                              |
| STA 391                  | 8D 91 03                      |                                                                                                                              |
| LDA 800, X               | BD 00 08                      |                                                                                                                              |
| INX                      | E8                            |                                                                                                                              |
| STA 800, X               | 9D 00 08                      |                                                                                                                              |
| DEX                      | CA                            |                                                                                                                              |
| LDA 391                  | AD 91 03                      |                                                                                                                              |
| STA 800, X               | 9D 00 08                      |                                                                                                                              |
| LDA #01                  | A9 01                         |                                                                                                                              |
| STA 390                  | 8D 90 03                      |                                                                                                                              |
| INX                      | E8                            |                                                                                                                              |
| CPX 800                  | EC 00 08                      |                                                                                                                              |
| BNE ?                    | D0 ?                          |                                                                                                                              |
| LDA 390                  | AD 90 03                      |                                                                                                                              |
| BNE ?                    | D0 ?                          |                                                                                                                              |
| RTS                      | 60                            |                                                                                                                              |

The assembly language instructions were copied directly from the flowchart in the order in which they appeared on the chart.

The operands for the branch commands will not be known until we actually start entering the machine language codes into the monitor.

We are now ready to begin typing the machine language program into the monitor. We will use (00's) to hold the place of the ?'s that are the operands of the relative addressing mode branch commands. When the entire program is finally entered and we can see the actual destination addresses, then we will replace the (00's) by the actual operands. To get the most from this discussion, you should follow along by actually entering this program into your Apple's memory. Enter the monitor by typing CALL -151 and begin to enter the program:



```

*300:A9 00 8D 90 03 A2 01 E8
*308:BD 00 08 CA DD 00 08 10
*310:(00) E8 BD 00 08 CA 8D 91
*318:03 BD 00 08 E8 9D 00 08
*320:CA AD 91 03 9D 00 08 A9
*328:01 8D 90 03 E8 EC 00 08
*330:D0 (00) AD 90 03 D0 (00) 60

```

(In this discussion, it is understood that the RETURN key must be hit at the end of each line entered into the monitor.)  
You should now type

\*300L

to get a listing of your program. This will help you to catch any typing errors and will also help to find the three missing operands for the branch commands.

The listing will appear as

```

0300- A9 00      LDA #$00
0302- 8D 90 03   STA $0390
0305- A2 01      LDX #$01
0307- E8         INX
0308- BD 00 08   LDA $0800,X
030B- CA         DEX
030C- DD 00 08   CMP $0800,X
030F- 10 00 1B   BPL $0311
0311- E8         INX
0312- BD 00 08   LDA $0800,X
0315- CA         DEX
0316- 8D 91 03   STA $0391
0319- BD 00 08   LDA $0800,X
031C- E8         INX
031D- 9D 00 08   STA $0800,X
0320- CA         DEX
0321- AD 91 03   LDA $0391
0324- 9D 00 08   STA $0800,X
0327- A9 01      LDA #$01
0329- 8D 90 03   STA $0390
032C- E8         INX
032D- EC 00 08   CPX $0800
0330- D0 00 05   BNE $0332
0332- AD 90 03   LDA $0390
0335- D0 00 09   BNE $0337
0337- 60         RTS

```

Type  
\*L to  
list the  
rest of  
the  
program

Ignore these destination  
addresses; they correspond  
to the placeholder operands

(00)

331

332

333



Compare this listing with the foregoing rough assembly language version. We can see now that the destination address of the BPL command is 032C. Therefore the operand of BPL is

Destination address—  $\frac{\text{Address of command}}{\text{following BPL}} = 032C - 0311 = \$1B$

You should type

\*310:1B

to replace the 00 placeholder by the true operand 1B. We next see that the real destination address of the first BNE command is \$307. The number of backward steps required is

$\frac{\text{Address of command} - \text{Destination address}}{\text{following BNE}} = 332 - 307 = 2B$

If you type

\*0-2B

you find the corresponding operand is D5. You should type

\*331:D5

to replace the 00 placeholder by the correct operand. Finally we see that the destination address of the second BNE command is location 300. The number of backward steps required is  $337 - 300 = 37$ . The corresponding operand is found by typing

\*0-37

and the result is C9. Type

\*336:C9

to enter the correct operand. You should now list the program again to make sure all went well. Let us run our program using the example list from the beginning of this section. Recall that the list is to be stored beginning in location \$801. The length of the list is to be stored in location 800. To sort the list 4, 6, 8, 2, 7 in ascending order, type

\*800:05 04 06 08 02 07

and then

\*300G



The sorted list will be stored in locations 801 to 805. Type

```
*801.805
```

to examine this memory range, with the result

```
0801- 02 04 06 07 08
```

We close the chapter by noting a few limitations of this sort algorithm. First, if you test this program, make sure all the numbers on the list to be sorted are between \$00 and \$7F inclusive. HEX numbers larger than \$7F cause problems because branch instructions interpret numbers between \$80 and \$FF as being negative. For example, consider how the sequence of steps

```
307- INX
308- LDA $0800,X
30B- DEX
30C- CMP $0800,X
30F- BPL $32C
```

Statements that switched a pair of  
elements that were in the wrong order  
were put here.

would operate on the numbers 05 and 90 if they were stored in locations 801 and 802. When  $X = 1$ , the expression

$$\text{Contents of } \$802 - \text{Contents of } \$801 = 90_{16} - 05_{16} = 8B_{16}$$

is evaluated by the CMP command. Since \$8B is negative, the BPL command would *not* jump over the section of the program that switches the contents of the storage locations. Thus the contents of 801 and 802 would be switched even though we want them left alone (because, as far as we are concerned, 05 is less than 90). It turns out that this problem with negative numbers can lead to very strange results in the sorted list. For instance, the order of elements in the final sorted list can sometimes depend on the order in which you type in the original unsorted elements! Without going into details here, let us just say it is wise to avoid numbers larger than \$7F when you test out the sort algorithm.

The reader will also notice that a loop to print out the final sorted list has not been included. This was done to keep the program as simple as possible. You may wish to modify the sort program to print the final results so that you do not have to use the monitor to view the sorted list after the program is run.

Finally, it should be noted that the longest list you can sort is 255 elements long, since in absolute indexed addressing the "subscript X" can only go as high as \$FF.

As you progress and become a skilled machine language programmer, you will find ways to overcome all the limitations mentioned here. For now, you should accept the program with its limitations. You might find it interesting to run the sort program on a list

of, say, 100 elements. Compare the execution time of the machine language bubble sort with the analogous version in BASIC. You will be amazed at how fast the machine language version runs compared with the BASIC version. It should be remembered that the main advantage of machine language is its extremely fast execution time. For this reason, the sorting problem is an ideal application of machine language programming. In the next chapter, we give another sort algorithm and another application of machine language programming—production of “Apple music.”

### Review Questions

1. What conditional branch command operand can be interpreted as meaning “take twelve steps backward”? \$\_\_\_\_\_
2. Use the monitor to perform the subtraction:  $509 - 1FD = \$$ \_\_\_\_\_
3. Give the DOS command you would use to BSAVE a binary program called LOOP. The program is stored in memory locations \$300 to \$370 (inclusive).
4. The command `CMP #09` causes which expression to be evaluated?
  - (a) contents of accumulator - 09
  - (b) 09 - contents of accumulator
5. BEQ is a \_\_\_\_\_ addressing mode command.
6. In the program segment

```
370:AD 90 03 CD 95 03 D0 ??
378:4C 50 03 (20) DA FD . . .
```

what operand should replace ?? if a branch to the circled byte is to be made when the result of the `CMP 395` calculation is not zero?

7. In the program segment

```
350:8D 05 03 88 F0 ?? . . .
```

the BEQ command is supposed to branch to location \$328 when the result of the DEY command is zero. What operand should replace the question marks?

8. 300— LDY #07  
302— body of the loop

```
DEY
BNE $302
```

How many times will the body of the loop be executed in this program segment?  
\$\_\_\_\_\_



9. 300- LDX #00  
302 body of the loop

INX  
CPX #0A  
BMI \$302

How many times will the body of the loop be executed in this program segment?

\$\_\_\_\_\_

10. 300- LDX #01  
302 body of the loop

INX  
CPX #08  
BNE \$302

How many times will the body of the loop be executed in this program segment?

\$\_\_\_\_\_

11. The assembly language command JMP closely resembles the BASIC command \_\_\_\_\_.
12. What advantage do relative addressing mode branch commands have over the absolute addressing mode JMP command?
13. The assembly language command \_\_\_\_\_ closely resembles the BASIC command GOSUB.

#### True or False

14. If you add  $80 + 80$  using the monitor, the result will be 00.
15. No result printed by the subroutine located at \$FDDA will ever contain a minus sign.
16. The command BLOAD can be used without the address parameter A.
17. If the result of a calculation is between \$80 and \$FF, then the N bit of the processor status register is set equal to one.
18. The CMP command can be used only in the immediate addressing mode.
19. If the result of the previous calculation is zero, then the BNE command will transfer control to the command stored immediately following BNE's operand.
20. All relative addressing mode commands are 2 bytes long.
21. When a relative addressing mode command is listed in assembly language, the destination address is shown instead of the command's actual operand.

## 22. The program segment

```
.  
. .  
CMP 395  
BNE 355  
BEQ 355  
.
```

is valid. It is legal for two different branch commands to share the same compare command.

## 23. The program segment

```
.  
. .  
DEY  
BNE 328  
.
```

is invalid as a compare command must always come before a branch command.

## 24. BPL considers zero a "plus" number.

**Programming Problems**

1. Write a loop that stores 16 random numbers in the consecutive memory locations S390 to S39F. Scan this list using another loop and print out only the positive numbers on the list.
2. Store the ASCII codes for the letters of a five-letter word in locations 390 to 394 and the ASCII codes for the letters of another five-letter word in locations 395 to 399. Print the words on separate lines and in alphabetic order.
3. Use a loop to draw a diagonal line running from the lower left-hand corner to the upper right-hand corner of the screen. Accomplish this by plotting the 40 low-resolution points that lie on the line.
4. Write a program that counts in *decimal* from 0 to 99. Store the ASCII code for 0 in locations 390 and 391. After printing the two codes next to one another (do this by storing the ASCII codes in the text screen locations \$400 and \$401), increment location S391 and print the two ASCII codes again. Continue incrementing location 391. Print the two ASCII codes stored in locations 390 and 391 each time you increment location 391. After printing 09, reset the ASCII code stored in



location 391 to B0 and increment the code stored in 390 so that you can print 10. Continue in this way until 99 has been printed. The program should be terminated after printing 99.

5. Use loops to print the digits 1 to 9 in NORMAL mode on one line of the screen. Print the digits in INVERSE mode below the NORMAL digits.

Note: The reader who is having trouble debugging any of these programs should read the section on debugging in Chapter 8 for some helpful hints at this time.

## ***Chapter 5***

# ***Nested Loops and Apple Sounds***

In this chapter, example programs are given that illustrate the use of nested loops. One interesting application of nested loops is the production of various tones using Apple's speaker. No new commands are introduced in this chapter; we extend some old ideas to new situations.

### **Example 1. Introduction to Nested Loops**

Consider the BASIC program

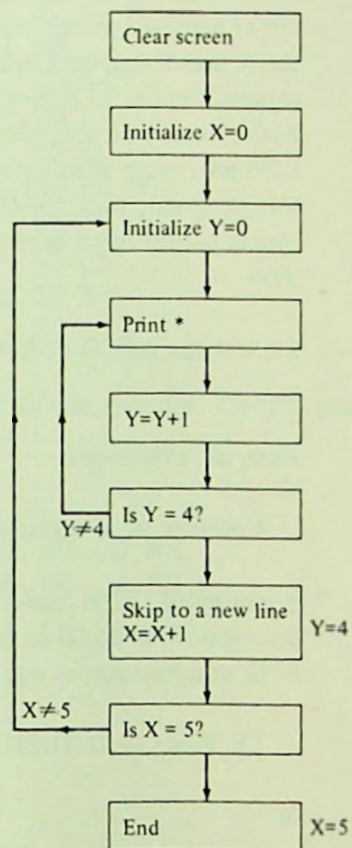
```
5 HOME
10 FOR X=1 TO 5
20 FOR Y=1 TO 4
30 PRINT "*";
40 NEXT Y
50 PRINT
60 NEXT X
```

The output is

```
****
****
****
****
****
```



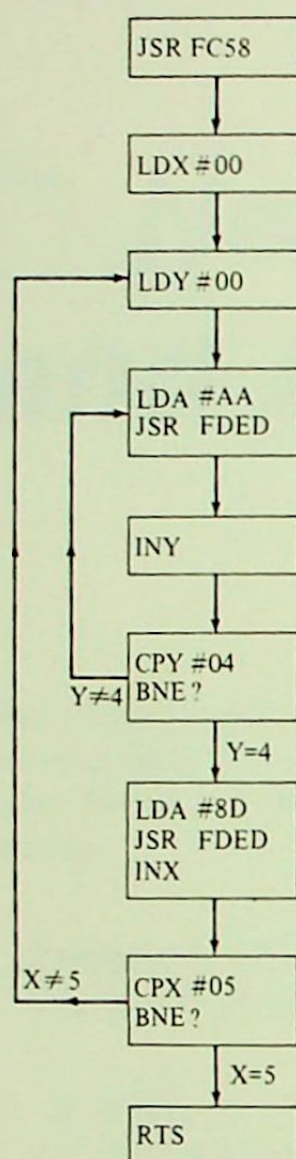
This program is a very simple example of nested loops. We can construct a flowchart corresponding to this program and then directly translate the flowchart into a machine language program.



Each block of this assembly language program is obtained by directly translating the flowchart into assembly language.

## Assembly Language

## Comments



By initializing  $Y = 0$  and placing the INY statement after the JSR FDED statement, the value stored in Y becomes equal to the number of asterisks printed so far. Setting up both the X and Y registers as counters in this way makes it easier to follow the program logic. Make sure you see why if LDX #01 and LDY #01 were used to initialize X and Y in this program, then the CPX and CPY statements would have been CPY #05 and CPX #06.

SAA is the ASCII code for the asterisk.

Here the expression

Contents of Y register - 04

is evaluated. BNE causes a branch if the expression is not equal to zero. Thus the CPY-BNE combination is equivalent to

IF  $Y-4 \neq 0$  THEN branch

or

IF  $Y \neq 4$  THEN branch

The operand of each BNE command is represented by a question mark. The actual operand will be filled in when the program is entered using the monitor.

The assembly language version is obtained by copying, in order, the commands given in the flowchart. The machine language codes are found with the aid of Appendix B.



| Assembly Language | Machine Language |
|-------------------|------------------|
| JSR FC58          | 20 58 FC         |
| LDX #00           | A2 00            |
| LDY #00           | A0 00            |
| LDA #AA           | A9 AA            |
| JSR FDED          | 20 ED FD         |
| INY               | C8               |
| CPY #04           | C0 04            |
| BNE ?             | D0 ?             |
| LDA #8D           | A9 8D            |
| JSR FDED          | 20 ED FD         |
| INX               | E8               |
| CPX #05           | E0 05            |
| BNE ?             | D0 ?             |
| RTS               | 60               |

Next, enter the program by use of the monitor. Use (00) placeholders for the unknown operands of BNE.

```
]CALL -151
*300:20 58 FC A2 00 A0 00 A9
*308:AA 20 ED FD C8 C0 04 D0
*310:(00) A9 8D 20 ED FD E8 E0
*318:05 D0 (00) 60
```

Get a listing by typing

```
*300L
```

```
0300- 20 58 FC    JSR $FC58
0303- A2 00      LDX #$00
0305- A0 00      LDY #$00
0307- A9 AA      LDA #$AA
0309- 20 ED FD    JSR $FDED
030C- C8         INY
030D- C0 04      CPY #$04
030F- D0 00      BNE $0311
0311- A9 8D      LDA #$8D
0313- 20 ED FD    JSR $FDED
0316- E8         INX
0317- E0 05      CPX #$05
0319- D0 00      BNE $031B
031B- 60         RTS
```

Ignore these addresses since they correspond to the placeholder operands 00

Ignore the rest of the listing; it is "garbage."

Now we can see the destination address of the first BNE command is 307, so that the number of backward steps required here is

$$\begin{aligned} \text{Address of command following BNE-Destination address} \\ = 311 - 307 = 0A \end{aligned}$$

The operand is found by typing

\*0-0A

The result is F6. Type

\*310:F6

to enter the correct operand to replace the 00 placeholder. Similarly the destination address for the second BNE command is \$305. Thus the number of backward steps required is  $31B-305 = \$16$  and the corresponding operand is EA. Type

\*31A:EA

to enter the correct operand for the second BNE command.

To run the program, type \*300G. The output will be as expected:

```
****
****
****
****
****
```

### Using the Mini-assembler

From the example just discussed and the bubble sort example of the last chapter, you can see that the major steps needed to produce a machine language program are as follows:

1. Construct a flowchart.
2. Write down the assembly language (mnemonic) instructions corresponding to each block of the flowchart. At this point, you cannot specify branch command operands.
3. Convert to machine language by looking up the op codes for each instruction. Put placeholders in for unknown branch command operands.
4. Using the List feature of the monitor, list the program. An assembly-type listing, complete with addresses for each command, will be produced. With the aid of this



listing, the correct operands for each branch command can be found. The monitor can be used to replace placeholder operands with the correct operands.

The mini-assembler allows you to avoid looking up op codes as in step 3. To enter a machine language program, the mnemonic labels in step 2 are typed directly. The mini-assembler program automatically converts the mnemonic codes into the machine language codes (this process is known as *assembling* the machine language program). In some cases, it even calculates operands for branch commands automatically.

The big problem with the mini-assembler is its lack of editing capabilities. Once a program has been completely entered, you cannot add or delete commands between two already existing commands by using the mini-assembler. However, you can use the monitor to edit your program as described in Chapter 2. The lack of editing capabilities is the reason for the name *mini-assembler*. A full-blown assembler has editing capabilities.

The mini-assembler is available in the standard Apple II only. If you have an Apple II Plus, you will need an Integer BASIC card or a diskette on which the mini-assembler program is stored (this diskette version of the mini-assembler is available commercially), if you wish to use the mini-assembler.

Let us use the mini-assembler to convert the mnemonic code of Example 1 into a machine language code. This mnemonic code is repeated here.

```

      JSR FC58
      LDX #00
      LDY #00
      LDA #AA
      JSR FDED
      INY
      CPY #04
      BNE ?
      LDA #8D
      JSR FDED
      INX
      CPX #05
      BNE ?
      RTS

```

First we need to enter the mini-assembler program. To do this from the monitor, type

\*F666G

or BRUN your mini-assembler if it is on a diskette. The mini-assembler will greet you with its prompt character (the exclamation point) and a beep from the speaker.

Next we need to give the address of the first command in our program, which normally will be \$300. You do not have to type \$'s or leading zeros in addresses. To enter the first command of our program, simply type the following: (The mini-assembler will display the exclamation point as a prompt. You should type everything else exactly as shown.)

```
!300:JSR FC58
```

In this discussion, it is assumed RETURN is hit after entering each command. Notice that the address SFC58 is *not* entered backward when using the mini-assembler. The reversal of low- and high-order bytes of the address is done automatically. As soon as you hit the RETURN key after entering a command, what you type disappears. After entering the JSR FC58 command, you will see

```
!0300- 20 58 FC      JSR $FC58
```

on the screen. Thus the listing of the program is produced line by line as you type in the commands, one at a time. To enter the second command, you do not have to type

```
!303:LDX #00
```

The mini-assembler automatically figures out the addresses of each command once the starting address has been specified for the first command. All you need to do to enter the second command is to type

```
! LDX #00
```



The space is required

If you omit the space between the exclamation point and the mnemonic label, the mini-assembler will beep the speaker and respond with the output

```
!LDX #00
^
```

The caret symbol (^) is used by the mini-assembler to point to errors. Any incorrect use of syntax, omitted operands, misspelled mnemonic codes, or other errors will result in a caret being displayed under or near the first incorrect character. If you do make an error, simply retype the command correctly.

Note that the # sign is an essential feature of the LDX #00 command. It tells the mini-assembler to select the immediate addressing mode. If the # sign is left out, the mini-assembler will assume the operand is an address and the zero-page addressing mode will be selected. (The zero-page addressing mode can be thought of as a special case of absolute addressing mode. If an address less than \$100 appears as an operand of any absolute addressing mode instruction, the zero-page op code automatically is selected by the mini-assembler.) Other addressing modes are symbolized using the standard assembly language conventions. Thus, if LDA 800, X were typed into the mini-assembler, the absolute indexed addressing mode (with index X) would be selected. No special symbols



are required when using implied mode commands. Merely type in the three-letter mnemonic code (INX, RTS, etc.) and the mini-assembler will correctly interpret the command as being in the implied mode.

After typing in the first seven instructions using the mini-assembler, the following will be displayed on the screen.

```
0300- 20 58 FC   JSR $FC58
0303- A2 00      LDX #$00
0305- A0 00      LDY #$00
0307- A9 AA      LDA #$AA
0309- 20 ED FD   JSR $FDED
030C- C8         INY
030D- C0 04      CPY #$04
!
```

The third through seventh instructions were entered exactly as the second instruction was. To enter the branch command, type

```
! BNE 307
```

and the line

```
030F- D0 F6      BNE $0307
```

will appear on the screen. Notice that the destination address for the branch command is typed in when the mini-assembler is used. As you can see, the real operand F6 was computed for you by the mini-assembler. It was especially easy to enter the branch command here because we were making a backward jump. Since the mini-assembler displays the assembly language and machine language code corresponding to each command right after it is typed in, the address of BNE's destination command (the LDA #AA command) was already displayed by the time you were ready to enter the BNE command into memory. In general, because the mini-assembler produces a listing of all commands entered so far, we can handle backward jumps directly. If the destination command of a branch instruction is a command not yet processed by the mini-assembler (as will be the case if the branch command jumps forward), then you will have to enter a placeholder address when entering the branch command. Forward branches cannot be entered directly when using the mini-assembler, for the same reason that they could not be entered directly using the monitor—the address of the branch's destination is not known at the time the branch command is to be entered. After you finish typing the entire program, you can go back to the monitor program, calculate the proper operand for the forward jump, and enter it using the monitor. Since this example program has two backward jumps, entering it is easy. If the rest of the program is entered, the result will be

```

0300- 20 58 FC      JSR $FC58
0303- A2 00         LDX #$00
0305- A0 00         LDY #$00
0307- A9 AA         LDA #$AA
0309- 20 ED FD      JSR $FDED
030C- C8            INY
030D- C0 04         CPY #$04
030F- D0 F6         BNE $0307
0311- A9 8D         LDA #$8D
0313- 20 ED FD      JSR $FDED
0316- E8            INX
0317- E0 05         CPX #$05
0319- D0 EA         BNE $0305
031B- 60            RTS

```

!

This is the same listing as we had after completing the last step in Example 1. The difference between using the mini-assembler and the method of Example 1 is that the finished product is much easier to obtain when the mini-assembler is used to produce it.

At this stage, you probably will want to leave the mini-assembler. To do this, you can type

```
!$FF69G
```

↙  
no space here

and the monitor prompt (\*) will appear. In the \$FF69G command, notice that there is no space between the exclamation point and the dollar sign. Although the dollar sign is optional most of the time in the mini-assembler, it is required here. A dollar sign after the exclamation point tells the mini-assembler that what comes after the dollar sign is a monitor command. Thus

```
!$FF69G
```

is equivalent to

```
*FF69G
```

The use of the \$ makes it possible to execute monitor commands without leaving the mini-assembler program. Since FF69 is the location of the monitor program, FF69G runs the monitor program and thus gets you out of the mini-assembler program. The \$ can be used to execute other monitor commands while still in the mini-assembler program. For example,



!\$300L would enable you to list a program stored beginning in location 300 without leaving the mini-assembler program,  
!\$300.330 would produce a memory dump of locations 300 to 330 inclusive,  
!\$800<300.310M would copy the contents of locations 300 to 310 (inclusive) into locations 800 to 810 (inclusive),

and so on.

It is recommended that you use the mini-assembler to enter your machine language programs from now on. It will make machine language programming a pleasanter experience.

### **Example 2. Keying the Inner Loop Index to the Outer Loop Index Value**

We next consider a problem in which the inner loop index's limits depend on the value of the outer loop index. This program illustrates a technique necessary for programming a type of sort algorithm that is even faster than the bubble sort. Consider the BASIC program

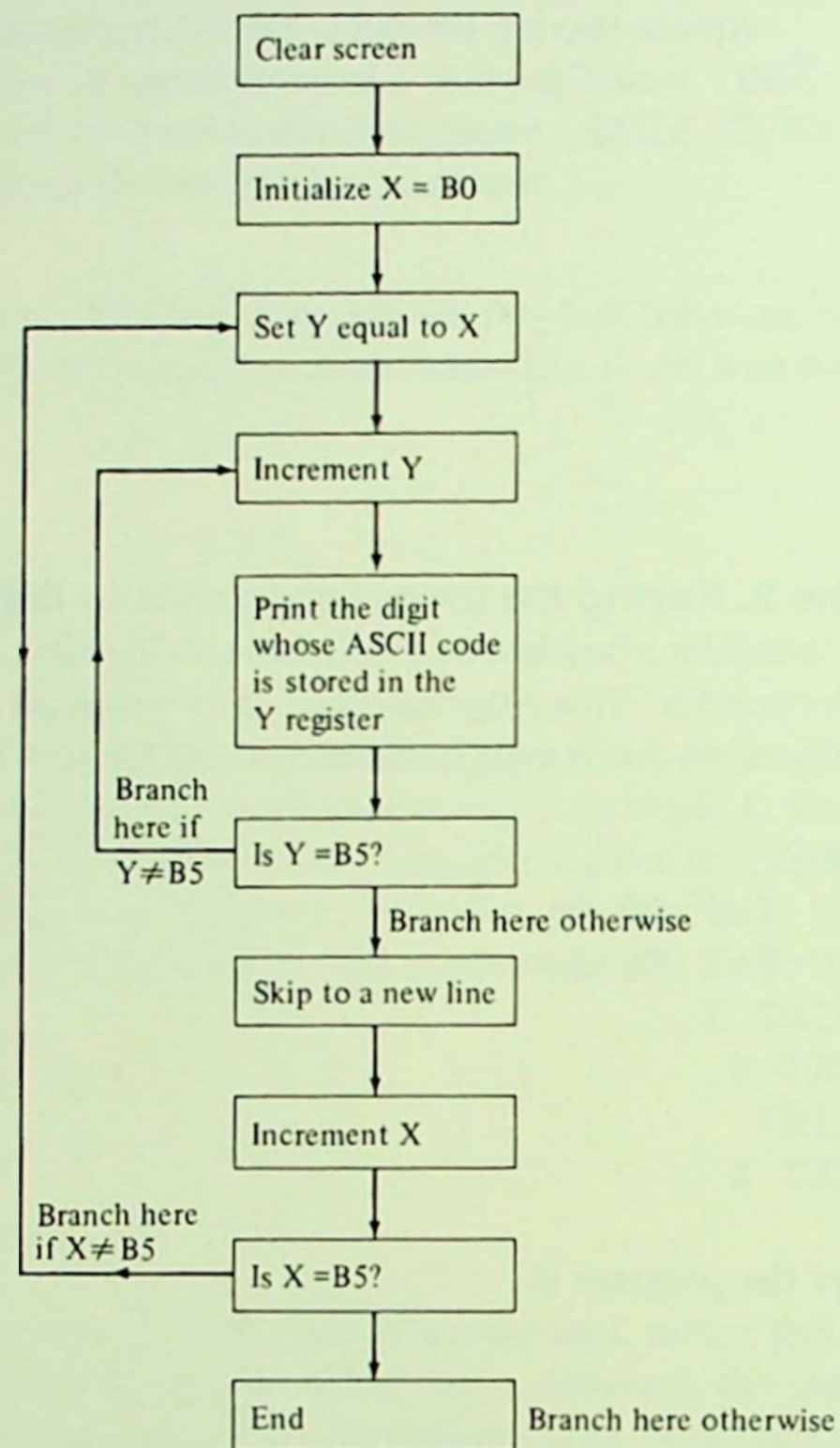
```
5 HOME
10 FOR X=1 TO 5
20 FOR Y=X TO 5
30 PRINT Y;
40 NEXT Y
50 PRINT
60 NEXT X
```

The output of the program is

```
12345
2345
345
45
5
```

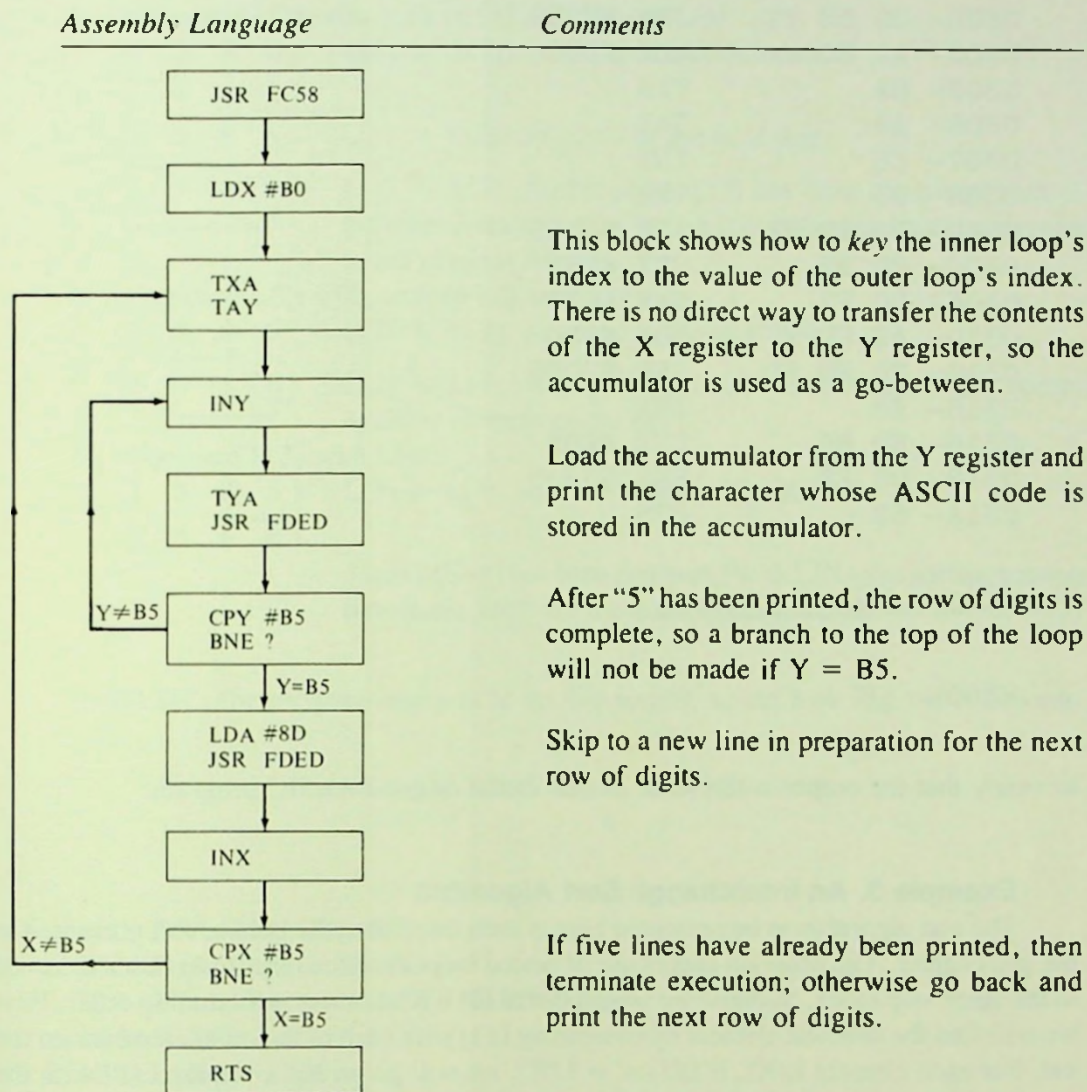
The distinguishing feature of the program is that the inner loop index is "keyed" to the outer loop index. This effect is fairly easy to duplicate in machine language. Before constructing the flowchart for this program, one difficulty needs to be overcome. If the "Print the Contents of the Accumulator" subroutine located at FDDA is used, a two-digit number is always printed. Thus this subroutine will not be used to print the digits 1 through 5 that appear in the output (if you tried, you would get the numbers 01 through 05). Instead we will use the "Print the Character Whose ASCII Code Is Stored in the Accumulator" subroutine located at FDED. If we use this routine, we will need to store the ASCII code for the digit in the accumulator, rather than the number itself. Thus, in the following program, the loop indexes take on the values B1 through B5 rather than 01 through 05.

## Flowchart



We now translate the flowchart into assembly language block by block.





With this flowchart, you can enter the machine language program directly into Apple's memory by using the mini-assembler. Note that both branch commands call for backward jumps, so the program can be entered in a very straightforward manner. All you have to do is type the mnemonic codes in the same order in which they appear in the blocks of the flowchart. When this is done, the final product will have this listing (you should go through the process of entering the program using your mini-assembler).

|       |          |            |
|-------|----------|------------|
| 0300- | 20 58 FC | JSR \$FC58 |
| 0303- | A2 B0    | LDX #\$B0  |
| 0305- | 8A       | TXA        |
| 0306- | A8       | TAY        |
| 0307- | C8       | INY        |
| 0308- | 98       | TYA        |
| 0309- | 20 ED FD | JSR \$FDED |
| 030C- | C0 B5    | CPY #\$B5  |
| 030E- | D0 F7    | BNE \$0307 |
| 0310- | A9 8D    | LDA #\$8D  |
| 0312- | 20 ED FD | JSR \$FDED |
| 0315- | E8       | INX        |
| 0316- | E0 B5    | CPX #\$B5  |
| 0318- | D0 EB    | BNE \$0305 |
| 031A- | 60       | RTS        |

You should run the program using

\*300G

to verify that the output is the same as that of the original BASIC program.

### Example 3. An Interchange Sort Algorithm

The sort algorithm to be presented here is even faster than the bubble sort presented in the last chapter. The program makes use of nested loops, and the inner loop index is keyed to the outer loop index. Suppose we want to sort a list  $L$  of numbers in ascending order. First we will find the smallest element by comparing  $L(1)$  with each of the other elements on the list. For each element  $L(X)$ , if  $L(1) \leq L(X)$ , we will go on and compare  $L(1)$  with the next element on the list,  $L(X+1)$ . On the other hand, if  $L(1) > L(X)$ , then we will switch the contents of  $L(1)$  and  $L(X)$ . After  $L(1)$  has been compared with each of the elements  $L(2)$ ,  $L(3)$ ,  $L(4)$ , . . . ,  $L(N)$  (where  $N$  is the number of elements in the list),  $L(1)$  will contain the smallest element of the original list. Next we will find the second smallest element in the list by comparing  $L(2)$  with each of the elements  $L(3)$ ,  $L(4)$ ,  $L(5)$ , . . . ,  $L(N)$ . You do not have to compare  $L(2)$  with  $L(1)$ .  $L(1)$  cannot possibly contain the second smallest element in the list because  $L(1)$  contains the smallest element in the list. If  $L(2) > L(X)$  for any  $X$ ,  $3 \leq X \leq N$ , then switch the contents of  $L(2)$  and  $L(X)$ . You would similarly find the third smallest element of the list by comparing  $L(3)$  with  $L(4)$ ,  $L(5)$ , . . . ,  $L(N)$ . The sorting process stops after you compare  $L(N-1)$  with  $L(N)$ , and make a switch if necessary.

The process is illustrated on this list: 5 2 8 6



1. Compare L(1) with each of L(2), L(3), and L(4).

$\begin{array}{cccc} 5 & 2 & 8 & 6 \end{array} \left. \vphantom{\begin{array}{cccc} 5 & 2 & 8 & 6 \end{array}} \right\} L(1) > L(2), \text{ so switch these elements.}$

$\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array} \left. \vphantom{\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array}} \right\}$

$\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array} \quad L(1) \leq L(3), \text{ so go on to the next pair.}$

$\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array} \quad L(1) \leq L(4). \text{ At this point, } L(1) \text{ has been compared with all the other elements in the list, so it contains the smallest element in the original list.}$

2. Compare L(2) with each of L(3) and L(4).

$\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array} \quad L(2) \leq L(3), \text{ so go on to the next pair.}$

$\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array} \quad L(2) \leq L(4). \text{ At this point, } L(2) \text{ must contain the second smallest element of the list.}$

3. Compare L(3) with L(4).

$\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array} \left. \vphantom{\begin{array}{cccc} 2 & 5 & 8 & 6 \end{array}} \right\} L(3) > L(4), \text{ so switch these elements.}$

$\begin{array}{cccc} 2 & 5 & 6 & 8 \end{array} \left. \vphantom{\begin{array}{cccc} 2 & 5 & 6 & 8 \end{array}} \right\}$

Since L(N-1) has been compared with L(N), the sorting process terminates with the list in ascending order.

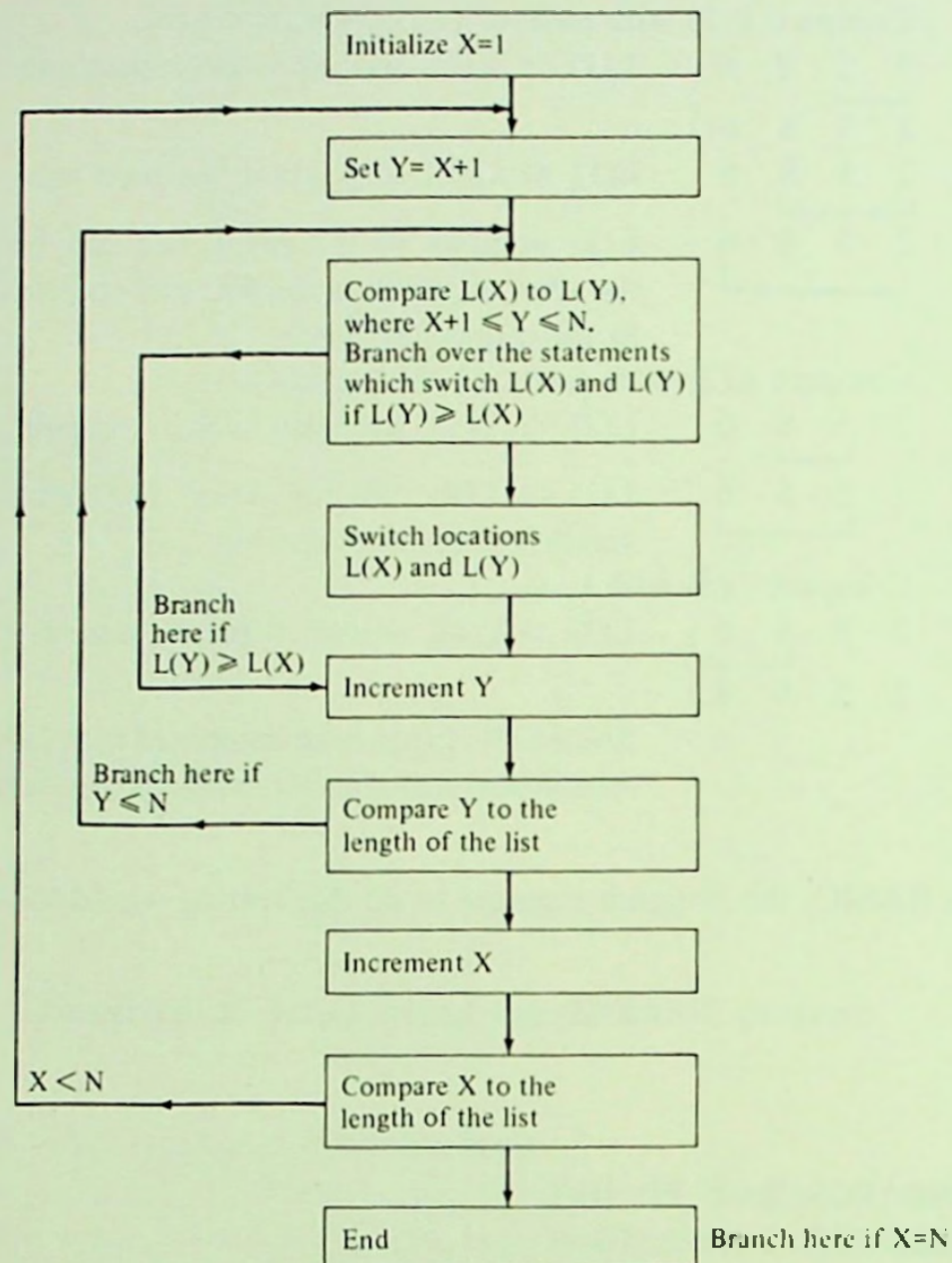
In BASIC, the program segment to do this sorting would look like the following:

```

      read in List L of N elements
      .
      .
      .
90 FOR X=1 TO N-1
100 FOR Y=X+1 TO N
110 IF L(Y) > = L(X) THEN 150
120 SAFE= L(X)
130 L(X)= L(Y)
140 L(Y)= SAFE
150 NEXT Y
160 NEXT X
      .
      .
      .

```

The flowchart for the sort algorithm is presented here. If you are still a little confused about the limits of the loop indexes (or "subscripts") X and Y, then study the BASIC program's loop limits and compare them with the limits used here.



We next convert the flowchart to assembly language block by block.

### Memory Use

Locations 801 to 800 + N will contain the List L of numbers to be sorted.

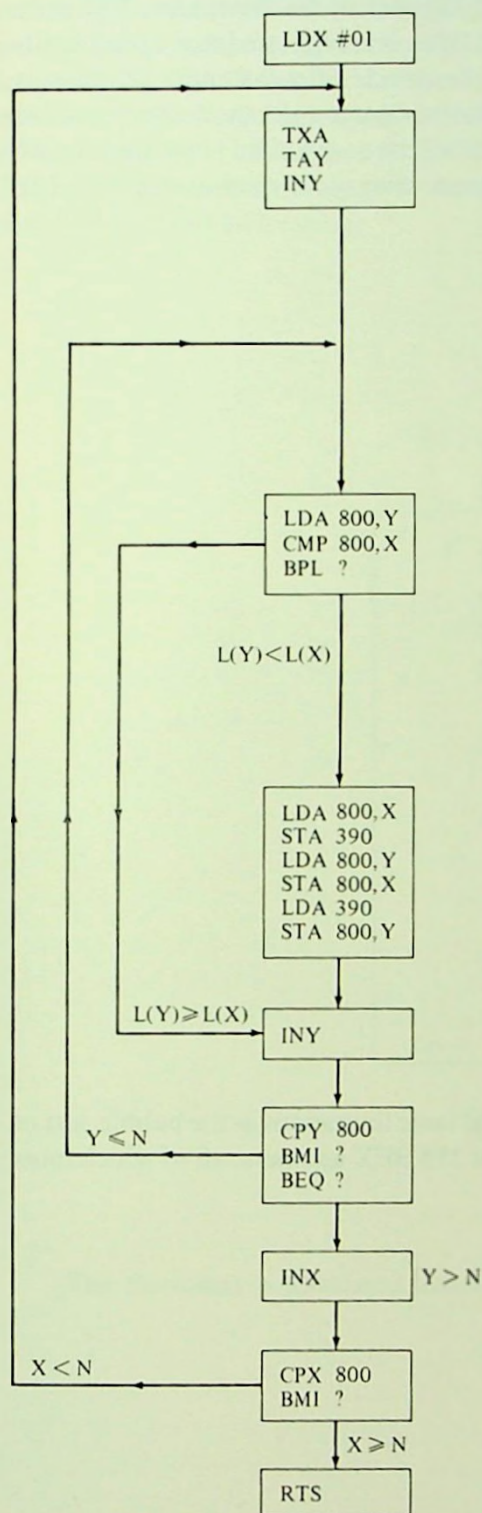
Location 800 will contain the length of the list (N).

Location 390 will be used as temporary storage when the contents of two memory locations have to be switched (analogous to "SAFE" in the BASIC program).



## Assembly Language

## Comments



X's value is transferred to the Y register using the accumulator as a go-between. Then Y is incremented. Therefore Y's value will be initialized to one more than X's value.

Notice that the sequence

```

LDA 800,X
CMP 800,Y
BMI ?

```

would also work. This sequence would be slightly less efficient than the one used here, however. If  $L(X) = L(Y)$ , then  $L(X) - L(Y)$  is not negative and a branch over the "switching instructions" would not occur if the alternate sequence was used. Of course, it is inefficient to "switch" elements if they are equal. The instructions given in the box make no such unnecessary switches.

These instructions switch elements  $L(X)$  and  $L(Y)$ .

The BMI command branches if  $Y < N$ . The BEQ command causes a branch if  $Y = N$ . The net result is that a branch will occur if  $Y \leq N$ .

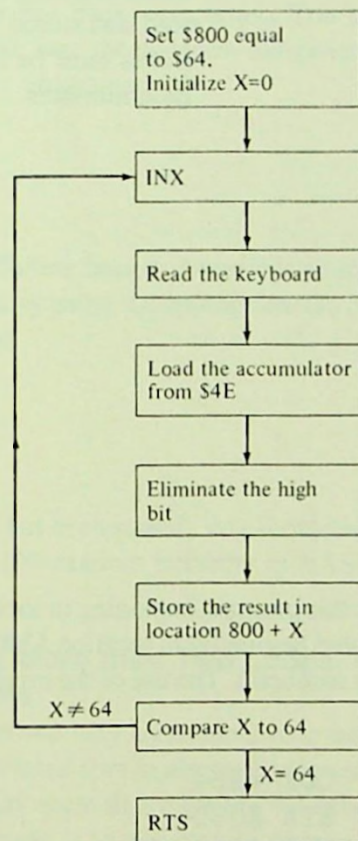
To enter this program in memory, we can type the assembly (mnemonic) codes, using the mini-assembler, in the same order in which they appear in the flowchart. The only problem will be finding the operand for the command BPL. It is suggested that a placeholder address such as \$300 be used when typing in the command. After the entire program is entered, you can go back and edit the program using the monitor to enter the correct operand for BPL. The operands of the other branch commands will be easy to find since they are all backward jumps. You should try entering the program using your mini-assembler. The listing of the finished product is

|       |          |              |
|-------|----------|--------------|
| 0300- | A2 01    | LDX #\$01    |
| 0302- | 8A       | TXA          |
| 0303- | A8       | TAY          |
| 0304- | C8       | INY          |
| 0305- | B9 00 08 | LDA \$0800,Y |
| 0308- | DD 00 08 | CMP \$0800,X |
| 030B- | 10 12    | BPL \$031F   |
| 030D- | BD 00 08 | LDA \$0800,X |
| 0310- | 8D 90 03 | STA \$0390   |
| 0313- | B9 00 08 | LDA \$0800,Y |
| 0316- | 9D 00 08 | STA \$0800,X |
| 0319- | AD 90 03 | LDA \$0390   |
| 031C- | 99 00 08 | STA \$0800,Y |
| 031F- | C8       | INY          |
| 0320- | CC 00 08 | CPY \$0800   |
| 0323- | 30 E0    | BMI \$0305   |
| 0325- | F0 DE    | BEQ \$0305   |
| 0327- | E8       | INX          |
| 0328- | EC 00 08 | CPX \$0800   |
| 032B- | 30 D5    | BMI \$0302   |
| 032D- | 60       | RTS          |

The sort algorithm presented here is subject to the same limitations as the bubble sort of the last chapter. The program will sort a maximum of 255 HEX numbers, all of which must be between \$00 and \$7F.



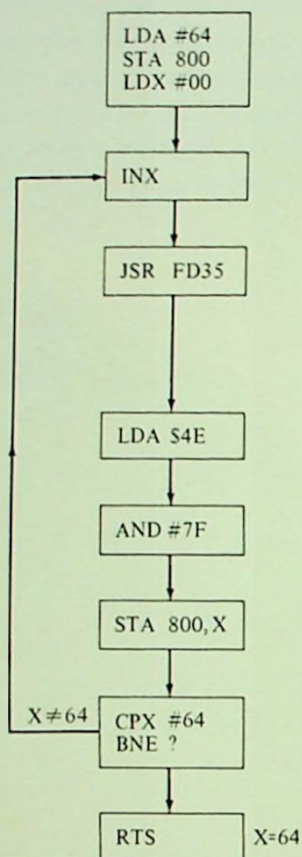
Let us write a program to generate 100 random HEX numbers between \$00 and \$7F and store the numbers in locations \$801 through \$864. We will then test the sort algorithm on this list of numbers. To accomplish the generation of random numbers, the subroutine located at \$FD35 (Read the Keyboard) will be used. Each time any key is hit, a random number will be stored in location \$4E. This number might be larger than \$7F, so it will be ANDed with 01111111 (binary) or \$7F to ensure that the leftmost (high) bit will be zero. The modified random number will then be stored in the appropriate memory location using absolute indexed addressing.

**Flowchart**

The flowchart is translated into assembly language here.

## Assembly Language

## Comments



Any key can be hit to generate each random number when the program is run. We do absolutely nothing with the ASCII code corresponding to the key actually hit. We are interested only in the random number generated and stored in \$4E. A total of 100 keystrokes must be made to generate all 100 random numbers.

Use the mini-assembler to enter this program beginning in location \$900 (\$900 is used since the sort algorithm is already stored beginning in location \$300, and locations \$800 to \$864 will be used to store the random numbers). The use of the mini-assembler is straightforward and yields

|       |          |              |
|-------|----------|--------------|
| 0900- | A9 64    | LDA #\$64    |
| 0902- | 8D 00 08 | STA \$0800   |
| 0905- | A2 00    | LDX #\$00    |
| 0907- | E8       | INX          |
| 0908- | 20 35 FD | JSR \$FD35   |
| 090B- | A5 4E    | LDA \$4E     |
| 090D- | 29 7F    | AND #\$7F    |
| 090F- | 9D 00 08 | STA \$0800,X |
| 0912- | E0 64    | CPX #\$64    |
| 0914- | D0 F1    | BNE \$0907   |
| 0916- | 60       | RTS          |



To generate the list of numbers to be sorted, type

```
*900G
```

A flashing cursor will prompt you to hit a key. Nothing much will happen when you do, since we did not have Apple print anything in this program. Each time you hit a key, a random number is being stored, however. Because of the way in which the subroutine \$FD35 works, you are going to have to hit 100 keys to generate your 100 random numbers. After you have struck the 100th key, monitor's prompt (\*) will reappear, informing you that execution of your program has been completed. You should examine the contents of locations 801 to 864 to make sure the program did generate 100 random numbers. After running the random number program, type

```
*300G
```

to run the sort algorithm. (Notice how it is possible to have two programs in memory at once. You can run either one by using the appropriate Go command.) Next examine the list using the monitor command

```
*801.864
```

to verify that the original list has been sorted. You should also note the speed with which the list was sorted (try sorting 100 random numbers in BASIC for comparison purposes).

#### **Example 4. Nesting More than Two Loops. Overcoming a Shortage of Registers**

The X and Y registers make very convenient loop indexes or general counters. They are also used to index *subscripted lists* in absolute indexed addressing. As your programs become more complex, it may seem that there are not enough registers to go around. One solution to the register shortage is to use regular storage locations as counters. Another solution is to use one register for more than one purpose. For example, suppose a program uses two subscripted lists. A regular memory location can be used to store the position, or *subscript*, of the element that needs to be worked with next in each list. Just before you work with the next element from the first list, load its subscript from memory (say location \$390) into the X register. Then you can use absolute indexed addressing to read the element from the first list. After working with the element from the first list, update the subscript stored in location \$390 so that Apple will "remember" which element in the first list to work with next. Just before you work with an element from the second list, load its subscript from memory (say location \$391) into the X register. Again absolute indexed addressing could be

used to select the appropriate element from the second list, once its subscript has been loaded into the X register. The value of the subscript stored in \$391 should be updated after working with the element from the second list. Note that when this method is used, the X register is not being "tied up" by storing either list's subscript during the entire program. The X register will contain a particular subscript only when it is actually needed so that many lists can be serviced by a single register. This technique can be extended from two lists to as many as you have in your program.

Consider the problem of producing the output

```
123123123123
123123123123
```

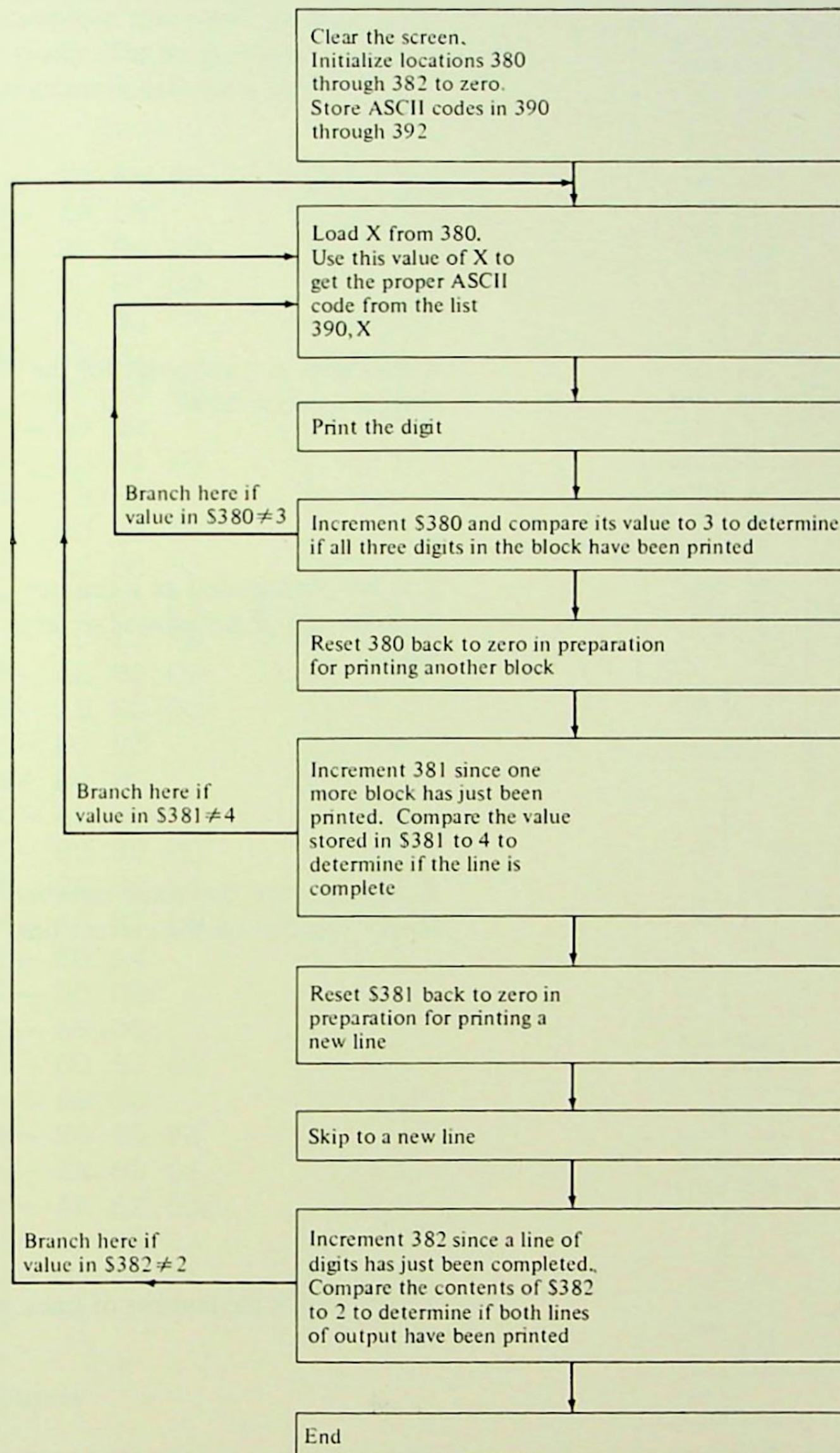
Suppose the ASCII codes for 1, 2, and 3 are stored in locations 390, 391, and 392. It would appear that one counter is needed to pick the proper ASCII code (on the list 390,X), another is needed to make sure only four "blocks" of 123's are printed on each line, and yet another counter is needed to make sure only two lines get printed. There are many ways to write the program that produces the foregoing output. The method presented here will employ only one register and the accumulator. Pay particular attention to how the utilization of regular memory locations allows you to use the X register to do several different jobs in the program.

### Memory Use

- 380 Keeps track of which ASCII code to print
- 381 Keeps track of the number of times the block 123 has been printed on each line
- 382 Keeps track of the number of lines printed
- 390 through 392 Contain B1, B2, B3 (ASCII codes for the digits)



## Flowchart

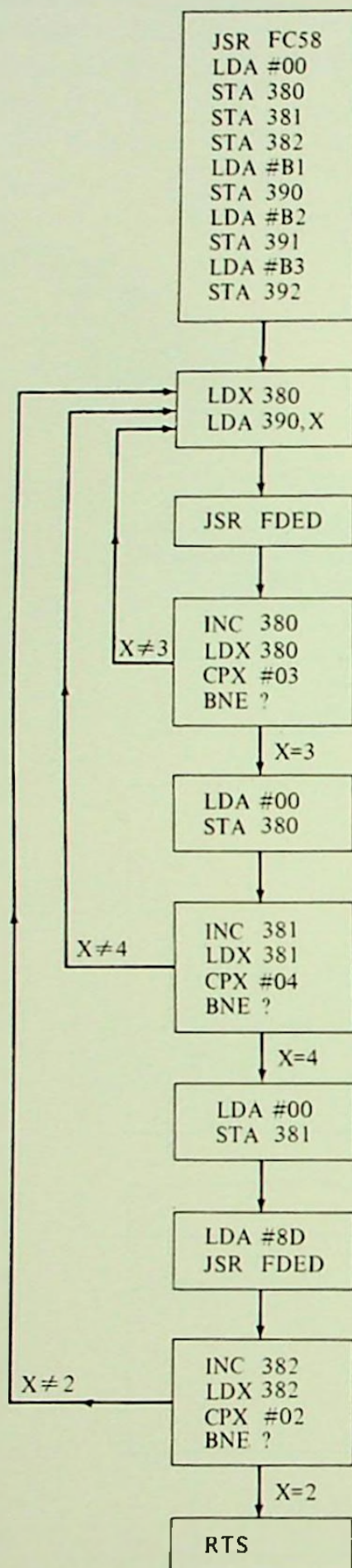


The assembly language version of the program is found by translating each block of the flowchart:



## Assembly Language

## Comments



X now acts as a subscript for the list beginning in location \$390.

X is now being used as a counter. X counts the number of digits printed so far in the "123 block."

X now counts the total number of "123 blocks" printed on the current line.

X now counts the number of lines printed so far.



The statements in the flowchart can be entered in the order in which they appear, using the mini-assembler. Since backward jumps only are involved, the complete program can be assembled easily. The result is rather lengthy for a program that does so little (keep in mind that this program is presented here to illustrate a technique, not because of its intrinsic value).

```
0300- 20 58 FC      JSR $FC58
0303- A9 00          LDA #$00
0305- 8D 80 03       STA $0380
0308- 8D 81 03       STA $0381
030B- 8D 82 03       STA $0382
030E- A9 B1          LDA #$B1
0310- 8D 90 03       STA $0390
0313- A9 B2          LDA #$B2
0315- 8D 91 03       STA $0391
0318- A9 B3          LDA #$B3
031A- 8D 92 03       STA $0392
031D- AE 80 03       LDX $0380
0320- BD 90 03       LDA $0390,X
0323- 20 ED FD       JSR $FDED
0326- EE 80 03       INC $0380
0329- AE 80 03       LDX $0380
032C- E0 03          CPX #$03
032E- D0 ED          BNE $031D
0330- A9 00          LDA #$00
0332- 8D 80 03       STA $0380
0335- EE 81 03       INC $0381
0338- AE 81 03       LDX $0381
033B- E0 04          CPX #$04
033D- D0 DE          BNE $031D
033F- A9 00          LDA #$00
0341- 8D 81 03       STA $0381
0344- A9 8D          LDA #$8D
0346- 20 ED FD       JSR $FDED
0349- EE 82 03       INC $0382
034C- AE 82 03       LDX $0382
034F- E0 02          CPX #$02
0351- D0 CA          BNE $031D
0353- 60             RTS
```

## Apple Sounds

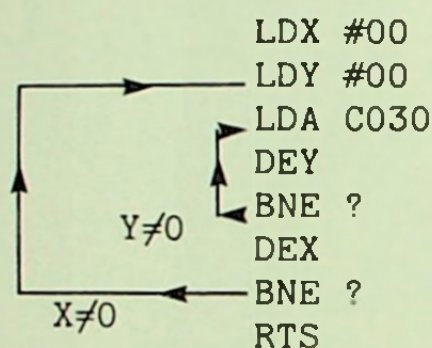
One of the more interesting uses of machine language is the production of various tones on Apple's speaker. By loading from the location \$C030, you can toggle or jiggle the speaker

once. By putting the command LDA \$C030 in a loop, you can cause the speaker to vibrate many times per second, producing musical tones. Obviously the larger the loop index limits, the longer the speaker will vibrate. Thus, by adjusting the loop index's upper limit, we can change the duration of our tones. By adjusting the time between speaker toggles, we can change the pitch of the tone. If the time between toggles (execution of the LDA \$C030 command) is short, then the speaker will vibrate rapidly, producing a high-pitched tone. Low tones are produced by increasing the time between successive speaker toggles.

The remaining examples in this chapter consist of increasingly sophisticated tone-producing programs. Although none of these programs can compare to commercially available music-composing programs, the essential elements needed to understand more complex programs are all here.

### Example 5. A First Attempt

#### Assembly Language



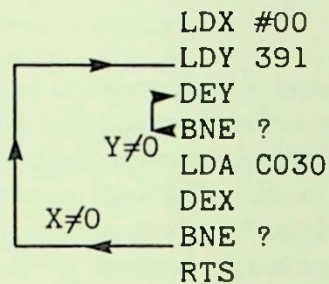
The inner (Y) loop rapidly toggles the speaker. The speaker will be toggled 256 times by the inner loop, since Y will have to take on all of the values \$FF, \$FE, \$FD, . . . , \$00 before the first branch command terminates the loop. The pitch of the note will be very high as there is almost no time between speaker toggles. The inner loop by itself does not last long enough to produce any sound to speak of, so the outer (X) loop executes the inner loop 256 times to produce a note that lasts long enough for you to hear it. Those who test the program barely will be able to hear the high-pitched note produced when it is run. The finished assembly listing for the program is as follows:

|       |          |            |
|-------|----------|------------|
| 0300- | A2 00    | LDX #\$00  |
| 0302- | A0 00    | LDY #\$00  |
| 0304- | AD 30 C0 | LDA \$C030 |
| 0307- | 88       | DEY        |
| 0308- | D0 FA    | BNE \$0304 |
| 030A- | CA       | DEX        |
| 030B- | D0 F5    | BNE \$0302 |
| 030D- | 60       | RTS        |



**Example 6. Varying the Pitch**

Because the LDA C030 instruction was inside the inner loop in Example 5, it was executed much too frequently. Consider this assembly language program:



The inner (Y) loop does nothing except stall for time. If the starting value of Y is large, then there will be a relatively large amount of time between speaker toggles. Thus the higher the value stored in \$391, the lower the pitch of the note (except that 00 gives the lowest pitch of all). The purpose of the X loop is to control the duration of the note. Since the starting value of X is set at 0, it is not possible to vary the duration of the note in this program.

One of the problems with this program is that the note's duration is much too short (the speaker is toggled only 256 times). We overcome this problem in the next example.

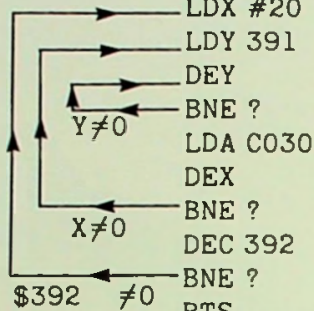
You should try running the program with different values stored in location 391 to get a feeling for the effect that this parameter has on the pitch. The program can be entered into memory using the mini-assembler. The finished product is shown here.

|       |          |            |
|-------|----------|------------|
| 0300- | A2 00    | LDX #\$00  |
| 0302- | AC 91 03 | LDY \$0391 |
| 0305- | 88       | DEY        |
| 0306- | D0 FD    | BNE \$0305 |
| 0308- | AD 30 C0 | LDA \$C030 |
| 030B- | CA       | DEX        |
| 030C- | D0 F4    | BNE \$0302 |
| 030E- | 60       | RTS        |

**Example 7. Varying the Pitch and Duration**

In Example 6, we had no control over the length of the tones. We also saw that the notes did not last very long. We can solve both of these problems if we take the entire program of Example 6 and put it inside a loop. Then the whole program automatically will be executed as many times as we wish, producing notes of any desired length. Consider the following assembly language program.

| Assembly Language | Memory Use |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDA 390           | 390        | The value you store 390 determines the length of the note. The higher the value, the longer the note (except that 00 gives the longest note of all).                                                                                                                                                                                                                                                                                                         |
| STA 392           |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| LDX #20           |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| LDY 391           |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| DEY               |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| BNE ?             | 392        | The value you store in location 390 is transferred to \$392. This is done because location \$392 is used as the counter that controls the length of the note. Had location 390 itself served this purpose, it would not have been possible to run the program twice in a row without resetting the value stored in 390. The original value stored in location \$390 would have been lost because DEC would have subtracted from it when the program was run. |
| LDA C030          |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| DEX               |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| BNE ?             |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| DEC 392           |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| BNE ?             |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| RTS               |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |



The main body of the program is the same as before, except for the LDX command, which now reads LDX #20 instead of LDX #00. The LDX #00 command results in the speaker being toggled 256 times in Example 6. In that example, the operand 00 was chosen to give the maximum possible duration for the tone. If the operand of LDX had not been changed in this example, then the duration of the notes could not have been controlled precisely enough. If the value stored in \$390 was N, then the speaker would have been toggled  $256_{10} * N$  times (since the X loop is executed N times when the value stored in \$390 is N). An increase of even one unit in the value stored in \$390 would increase the number of speaker toggles by 256. By changing the operand of LDX to \$20, an increase of one in the value stored in \$390 will increase the number of speaker toggles by only thirty-two. With this change, increases in N cause a smaller change in the number of speaker toggles than was the case before. This allows us to "fine tune" the length of the tone to a much greater degree than would have been possible had we used LDX #00.

You probably noticed one other feature of this sound program. Low-pitched notes last longer than high-pitched notes, even when identical values are stored in \$390. This is so because the Y loop, which controls pitch, also controls the length of the note to a certain degree. Low pitch is produced by increasing the time between speaker toggles. Since the number of times the speaker is toggled is the same for all notes (when the same value is stored in \$390), it follows that the low notes will last slightly longer than the high notes. If this is a problem, you can always compensate by storing slightly lower values in \$390 when low notes are played. When the program is entered using the mini-assembler, the following listing results.



```

0300- AD 90 03      LDA $0390
0303- 8D 92 03      STA $0392
0306- A2 20          LDX #$20
0308- AC 91 03      LDY $0391
030B- 88             DEY
030C- D0 FD          BNE $030B
030E- AD 30 C0      LDA $C030
0311- CA             DEX
0312- D0 F4          BNE $0308
0314- CE 92 03      DEC $0392
0317- D0 ED          BNE $0306
0319- 60             RTS

```

You should run the program using different values stored in \$390 and \$391 to get a feeling for the effect these parameters have on the tone produced by the program.

### Example 8. Playing More than One Note

We conclude this chapter with one last programming example. One limitation of all programs so far is that they play only one note. To play a tune, you would have to reset the pitch and duration for each note and type

```
*300G
```

to play the note. It is just not possible to do this fast enough to play a tune. In this example, we write a program that reads both the duration and pitch of notes from absolute indexed lists. The Apple automatically will play as many notes as we wish in rapid succession. This program will be written by placing the program of Example 7 inside a loop that reads the duration and pitch of each note. The note will be played using the program of Example 7. In writing the program of Example 7, the X register was used as a counter to help control the duration of the notes. In this program, a regular memory location will be used instead. This will free the X register so that it can be used in absolute indexed addressing commands that read the duration and pitch of each note to be played.

### Memory Use

- 391 The pitch of each note is transferred here after reading it from the list of pitches.
- 392 This counter controls the duration of each note (in conjunction with location \$393).
- 393 This location is used as a counter to help control the duration of each note. It does the same job that the X register did in Example 7.
- 800 The number of notes to be played is stored here (N).
- 801 to 800+N List containing the duration of each of the N notes is stored here.
- 831 to 830+N List containing the pitch of each of the N notes is stored here.
- X register This register is used as a subscript for the pitch and duration lists.

| Assembly Program | Comments                                                                                                                                                                          |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDX #00          |                                                                                                                                                                                   |
| INX              | The duration of the note is read and stored in location                                                                                                                           |
| LDA 800, X       | \$392.                                                                                                                                                                            |
| STA 392          |                                                                                                                                                                                   |
| LDA 830, X       | The pitch is read and stored in location \$391.                                                                                                                                   |
| STA 391          |                                                                                                                                                                                   |
| LDA #20          | The LDA #20 and STA 393 commands do the job that<br>LDX #20 did in the program of Example 7.                                                                                      |
| STA 393          |                                                                                                                                                                                   |
| LDY 391          | This is the program of Example 7 rewritten using \$393<br>to take the place of the X register.                                                                                    |
| DEY              |                                                                                                                                                                                   |
| BNE ?            |                                                                                                                                                                                   |
| Y ≠ 0            |                                                                                                                                                                                   |
| LDA C030         |                                                                                                                                                                                   |
| DEC 393          |                                                                                                                                                                                   |
| BNE ?            |                                                                                                                                                                                   |
| \$393 ≠ 0        |                                                                                                                                                                                   |
| DEC 392          |                                                                                                                                                                                   |
| BNE ?            |                                                                                                                                                                                   |
| \$392 ≠ 0        |                                                                                                                                                                                   |
| CPX 800          | The outer loop runs the program of Example 7 N times<br>(N is the number of notes to be played, as stored in<br>location \$800) automatically to play our sequence of N<br>notes. |
| BNE ?            |                                                                                                                                                                                   |
| X ≠ N            |                                                                                                                                                                                   |
| RTS              |                                                                                                                                                                                   |

The completed assembly listing of the program is

|       |          |               |
|-------|----------|---------------|
| 0300- | A2 00    | LDX #\$00     |
| 0302- | E8       | INX           |
| 0303- | BD 00 08 | LDA \$0800, X |
| 0306- | 8D 92 03 | STA \$0392    |
| 0309- | BD 30 08 | LDA \$0830, X |
| 030C- | 8D 91 03 | STA \$0391    |
| 030F- | A9 20    | LDA #\$20     |
| 0311- | 8D 93 03 | STA \$0393    |
| 0314- | AC 91 03 | LDY \$0391    |
| 0317- | 88       | DEY           |
| 0318- | D0 FD    | BNE \$0317    |
| 031A- | AD 30 C0 | LDA \$C030    |
| 031D- | CE 93 03 | DEC \$0393    |
| 0320- | D0 F2    | BNE \$0314    |
| 0322- | CE 92 03 | DEC \$0392    |
| 0325- | D0 E8    | BNE \$030F    |
| 0327- | EC 00 08 | CPX \$0800    |
| 032A- | D0 D6    | BNE \$0302    |
| 032C- | 60       | RTS           |



To run the program, first store the number of notes to be played in location \$800. The duration of each note is stored beginning in location \$801. The corresponding pitches are stored beginning in location \$831. A sample run might be made using

The 00 in \$830 is not used for anything since the first pitch is stored in \$831. This byte is entered so that the line can begin with location \$830. Now the pitches and durations are lined up on Apple's screen.

```
*800:0A 20 19 18 17 16 15 14
*808:13 12 11
*830:00 20 30 40 50 60 70 80
*838:90 A0 B0
*300G
```

### Review Questions

1. The mini-assembler's prompt symbol is the \_\_\_\_\_.
2. The assembly language command \_\_\_\_\_ means "load the accumulator with \$05," whereas the assembly language command \_\_\_\_\_ means "load the accumulator from location \$05."
3. You are entering a program using the mini-assembler. The following program segment is displayed on the screen.

```
0328- E8      INX
0329- E0      CPX #$09
```

- You wish to enter a BNE command after the CPX command, which will cause a conditional jump to location \$31A when the value stored in the X register is not \$09. To enter the BNE command, you would type \_\_\_\_\_.
4. To leave the mini-assembler program and return to the monitor, use the command \_\_\_\_\_.
  5. To obtain a program listing beginning with the command stored in location \$300 without leaving the mini-assembler program, use the command \_\_\_\_\_.
  6. The HEX number N is stored in the X register. Give the three assembly language commands needed to store  $N - 1$  in the Y register.
  7. In the sound program

```

0300- LDY #$10
0302- LDX $0351
0305- DEX
0306- BNE $0305
0308- LDA $C030
030B- DEY
030C- BNE $0302
030E- DEC $0350
0311- BNE $0300
0313- RTS

```

the value stored in location \$ \_\_\_\_\_ controls the length of the notes and the value stored in location \$ \_\_\_\_\_ controls the pitch of the notes.

### TRUE or FALSE

8. The mini-assembler lacks editing capabilities.
9. When you use the mini-assembler, it is necessary to give a storage address for each command you want to enter.
10. You must leave a space in front of all assembly language commands you store in memory using the mini-assembler.
11. The commands

```

LDA #01
JSR FDDA

```

will cause

1

to be printed on the screen.

12. The instructions

```

LDA 800,X
STA 390
INX
LDA 800,X
DEX
STA 800,X
LDA 390
INX
STA 800,X

```



will switch the contents of locations \$805 and \$806 when the value stored in the X register is \$05.

13. If you type

```
LDA 390,X
```

when using the mini-assembler, then the absolute indexed addressing mode op code for LDA will be selected by the mini-assembler.

### Programming Problems

1. Write a program that produces this output

```
1
12
123
1234
12345
```

2. Write a program that produces the following output

```
ABCDEABCDEABCDE
ABCDEABCDEABCDE
ABCDEABCDEABCDE
ABCDEABCDEABCDE
```

using only regular memory locations and the accumulator.

3. Suppose a list of nine numbers corresponding to nine pitches ranging from low to high is stored in memory. Write a program that enables you to select any of the nine notes from the keyboard. Striking one of the keys labeled "1" to "9" should cause the appropriate note to be played. The program should let users play as many notes as they wish. Terminate the program when the "S" (for STOP) key is struck.
4. Modify the sort algorithm presented in this chapter so that the original list is sorted in descending order instead of ascending order.
5. Write a machine language program to produce a list of all pairs of numbers X Y such that  $1 \leq X \leq 6$  and  $1 \leq Y \leq 6$ . Consider the pair X Y to be the same as the pair

Y X. Do not list any pair more than once and list each pair on a separate line. Your output should look as follows:

1 1

1 2

.

.

1 6

2 2

2 3

.

.

2 6

3 3

3 4

.

.



# Chapter 6

## Arithmetic

Machine language is not well suited for doing complex arithmetic computations. As we will see, all operations except addition and subtraction are difficult to carry out. BASIC should be used to run any program involving many arithmetic operations. In this chapter, the addition capabilities of machine language are stressed. Subtraction and multiplication also are examined.

### Adding Two HEX Numbers

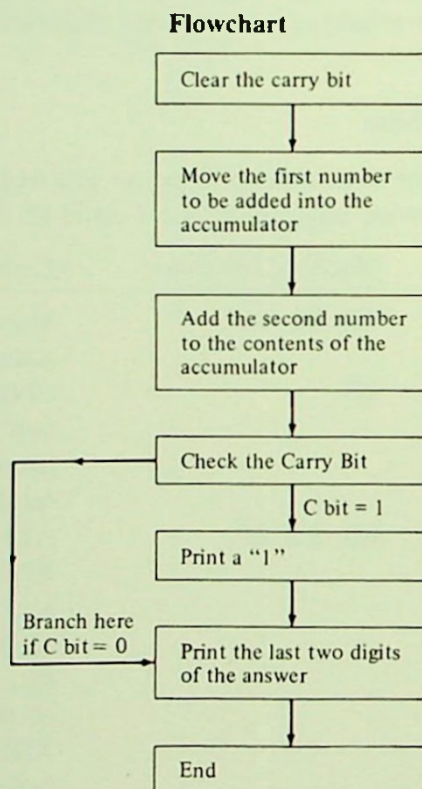
All addition takes place in the accumulator. Suppose you wish to add two numbers and display the result. The following program segment could be used.

| <i>Assembly Language</i> | <i>Machine Language</i> | <i>Comments</i>                                                                                                                                                                                                                                                |
|--------------------------|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDA 390                  | AD 90 03                | Move the first number to be added into the accumulator.                                                                                                                                                                                                        |
| CLC                      | 18                      | CLear the Carry bit. The function of this command is explained in detail in the next section.                                                                                                                                                                  |
| ADC 391                  | 6D 91 03                | Add the number stored in \$391 to the contents of the accumulator. The result is stored back into the accumulator, wiping out the number originally stored there. The ADC (ADd with Carry command) is being used in the absolute addressing mode (op code 6D). |
| JSR FDDA                 | 20 DA FD                | Display the result.                                                                                                                                                                                                                                            |

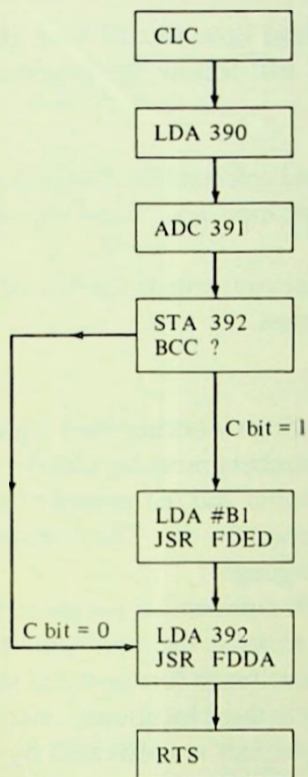
The ADC command can be used in any addressing mode studied thus far (except for the implied and relative addressing modes). See the table in Appendix B for the appropriate op codes.

The ADC command makes use of the C or Carry bit in the processor status register. Suppose we add the two HEX numbers \$90 and \$80. Because only two digits fit in any memory location, the result stored in the accumulator will not be \$110, but \$10. The leading one is not completely lost, however. If the result of an addition is greater than \$FF, then the Carry bit will be set to one. For reasons described in the next section, it is important that the Carry bit be zero at the beginning of any addition problem. The implied addressing mode command CLC (CLear the Carry Bit, op code 18) should come immediately before you do any addition. This command will put a zero in the carry bit.

There are two relative addressing mode branch commands that can be used to take advantage of the information stored in the Carry bit: BCC (Branch if the Carry Bit is Clear, op code 90) and BCS (Branch if the Carry bit is Set equal to 1, op code B0). Both commands use the Carry bit to determine whether branching will occur. BCC will branch if a zero is found in the C bit and BCS will branch if a one is found in the C bit. Consider the following modification to the four-line program segment given previously. This modification uses a branch command to print out the correct result for any addition problem whose answer is three digits long.





*Assembly Language**Comments*

Clear the Carry Bit should be the first step in any addition problem.

The first number to be added is stored in 390.

The second number to be added is stored in 391.

The sum of the two numbers is placed in \$392 for safekeeping. If a "1" has to be printed in front of the sum, we will need to load one's ASCII code into the accumulator. We store the sum in location \$392 now to avoid losing the sum later if the "1" does have to be printed. It should also be noted that the sub-routine located at FDDA will scramble the contents of the accumulator. Therefore it is important to store the result of any addition before it is printed, if the result will be needed later in the program.

The finished assembly listing for the program is

|       |          |            |
|-------|----------|------------|
| 0300- | 18       | CLC        |
| 0301- | AD 90 03 | LDA \$0390 |
| 0304- | 6D 91 03 | ADC \$0391 |
| 0307- | 8D 92 03 | STA \$0392 |
| 030A- | 90 05    | BCC \$0311 |
| 030C- | A9 B1    | LDA #\$B1  |
| 030E- | 20 ED FD | JSR \$FDED |
| 0311- | AD 92 03 | LDA \$0392 |
| 0314- | 20 DA FD | JSR \$FDDA |
| 0317- | 60       | RTS        |

### Adding HEX Numbers More than Two Digits Long

By studying this decimal addition example, we can understand how to add long HEX numbers. The flowchart used in machine language addition will follow the paper-and-pencil method of addition almost exactly.

|                                                             |                                                                                                                                                                                                      |
|-------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\begin{array}{r} ^128 \\ \underline{34} \\ 2 \end{array}$  | <p>STEP 1. Work with the pair of digits in the right-hand column first. Since <math>8 + 4</math> gives a two-digit answer, you write the last digit and "carry the one" over to the next column.</p> |
| $\begin{array}{r} ^128 \\ \underline{34} \\ 62 \end{array}$ | <p>STEP 2. Work with the next pair of digits. The answer you write is the sum of the two digits in the left column plus any carries.</p>                                                             |

Had there been more digits in the problem, you would have just kept working from right to left. To find the digit that gets written in each column, *three* numbers must be added—the two digits in the original problem that appear in the given column and the number being carried. The ADC command automatically takes care of carries for us. The following program will illustrate the treatment of carries in machine language.

The program to be presented here adds two four-digit HEX numbers. If we are to add two four-digit HEX numbers, we first have to decide how to store the numbers. It is convenient to store each number in consecutive memory locations (each location will store two digits of the numbers to be added). We will store the answer in three locations, since the sum conceivably could be five digits long. The following flowchart is motivated by the paper-and-pencil addition algorithm illustrated.

### Memory Use

390  
Most significant byte  
of first number to be  
added (first two digits)

392  
Most significant byte  
of second number to be  
added

394  
Most significant byte  
of the sum

396  
Least significant byte  
of the sum

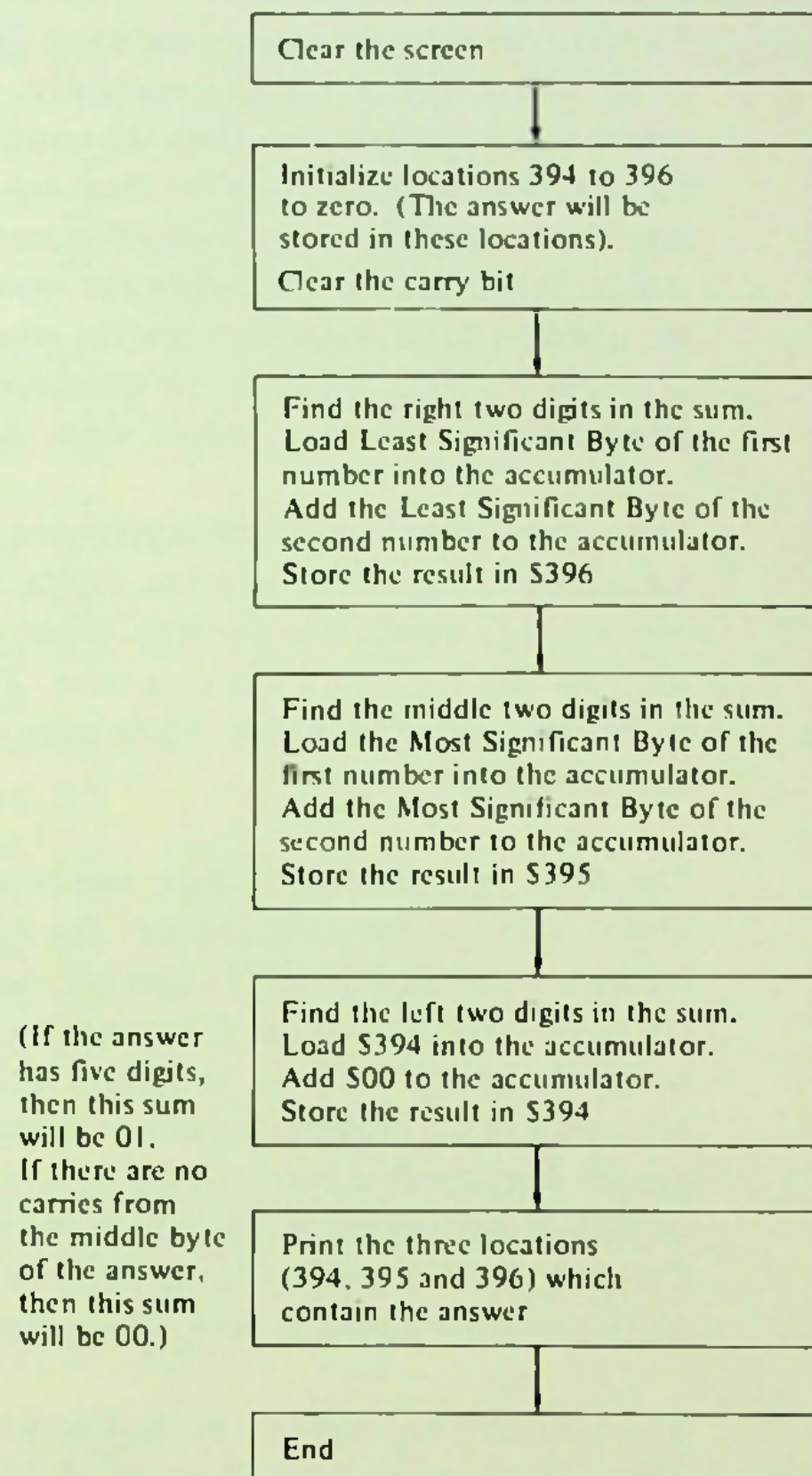
391  
Least significant byte  
of first number to be  
added (last two digits)

393  
Least significant byte  
of second number to be added

395  
Middle byte of  
the sum



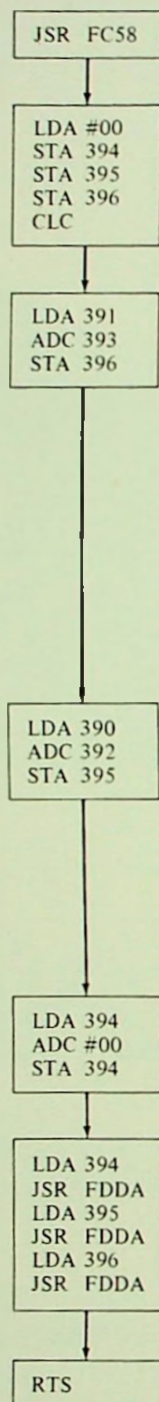
## Flowchart



---

Assembly Language      Comments

---



To make the program shorter and more flexible, the numbers to be added will be entered manually, using the monitor, before the program is run. Both the numbers to be added and the sum will be stored with the least significant bytes in the highest memory locations. Under this arrangement, the digits will be in the correct order when the monitor is used to enter the numbers to be added or when the sum is to be displayed. (Type in the HEX digits of the numbers to be added in the same order in which you would write them on paper.)

The contents of one memory location are added in one step to the contents of a second location. Apple is capable of adding two two-digit numbers in one step. When we add on paper, we add two one-digit numbers at a time. This is the only real difference between the way Apple adds and the way we add with pencil and paper.

The ADC command adds three numbers—the number presently stored in the accumulator, the number stored in the location mentioned in ADC's operand, and the contents of the Carry bit. Notice that the Carry bit is cleared only once—before doing any addition. The Carry bit has to be zero at the beginning of the problem because it is impossible to have any carries when the first bytes (on the right side of the numbers to be added) are processed. In each of the remaining steps, the Carry bit is not cleared so that any carries from previous steps will be properly added to the bytes currently being processed.

This step adds the contents of the Carry bit to location \$394. The value stored in 394 will be zero if the Carry bit was zero, and 01 if the Carry bit was set.



The finished assembly listing of this program would look like the following.

```

0300-20 58 FC      JSR $FC58
0303-A9 00         LDA #$00
0305-8D 94 03      STA $0394
0308-8D 95 03      STA $0395
030B-8D 96 03      STA $0396
030E-18           CLC
030F-AD 91 03      LDA $0391
0312-6D 93 03      ADC $0393
0315-8D 96 03      STA $0396
0318-AD 90 03      LDA $0390
031B-6D 92 03      ADC $0392
031E-8D 95 03      STA $0395
0321-AD 94 03      LDA $0394
0324-69 00         ADC #$00
0326-8D 94 03      STA $0394
0329-AD 94 03      LDA $0394
032C-20 DA FD      JSR $FDDA
032F-AD 95 03      LDA $0395
0332-20 DA FD      JSR $FDDA
0335-AD 96 03      LDA $0396
0338-20 DA FD      JSR $FDDA
033B-60           RTS

```

To add the HEX numbers \$F032 and \$3040 using this program, you would enter me two numbers using the monitor command

```
*390:F0 32 30 40
```

and run the program using

```
*300G
```

The result displayed would be

```
012072
```

### The Decimal Mode

Consider the addition problem

```

  08
+ 04


---



```

In HEX, the answer is 0C. It is possible to interpret this problem as a decimal addition problem, in which case the answer would be 12. Normally all arithmetic in machine language is done in HEX. There are times when it would be advantageous to interpret the digits stored in memory as representing decimal numbers rather than HEX numbers. This can be done using the implied mode command SED (*SEt the Decimal mode*, op code F8). In the decimal mode, addition and subtraction that occur in the accumulator are done using decimal addition and subtraction facts instead of HEX addition and subtraction facts. In other words, if Apple adds or subtracts, the answer will be given in decimal, not HEX. Notice that it would make no sense to add numbers such as A5 and 6D while in decimal mode. Consider this program:

| Assembly Language | Comments                                                                                                                                                                                                    |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LDA #00           |                                                                                                                                                                                                             |
| SED               |                                                                                                                                                                                                             |
| LDY #00           |                                                                                                                                                                                                             |
| CLC               | Each time the loop is executed, 03 is added to the accumulator. The Carry bit is cleared because the beginning of each pass through the loop is the beginning of a new addition problem.                    |
| Y≠6               |                                                                                                                                                                                                             |
| ADC #03           |                                                                                                                                                                                                             |
| INY               |                                                                                                                                                                                                             |
| CPY #06           |                                                                                                                                                                                                             |
| BNE ?             | Branch commands can affect the Carry bit. Because commands other than ADC can alter the Carry bit's contents, never assume the Carry bit is clear. Always use CLC at the beginning of any addition problem. |
| JSR FDDA          |                                                                                                                                                                                                             |
| RTS               |                                                                                                                                                                                                             |

This program is designed to add 03 to the accumulator 06 times. The result is 18, since the addition is performed in decimal mode. Had the decimal mode not been set, the result would have been \$12, since  $6 \times 3 = 12$ .

The complete assembly listing for the last program is included for the sake of completeness:

|       |          |            |                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-------|----------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0300- | A9 00    | LDA #\$00  |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 0302- | F8       | SED        |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 0303- | A0 00    | LDY #\$00  |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 0305- | 18       | CLC        |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 0306- | 69 03    | ADC #\$03  | Notice that even in the decimal mode, operands are displayed with a \$. This is so because all values in the program are still <i>stored</i> in HEX, but the decimal mode <i>interprets</i> numbers to be added as decimal numbers. For example, in decimal mode the number "12" is interpreted to mean "10 plus 2" rather than "16 plus 2" and decimal addition facts are used to add numbers rather than HEX addition facts. |
| 0308- | C8       | INY        |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 0309- | C0 06    | CPY #\$06  |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 030B- | D0 F8    | BNE \$0305 |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 030D- | 20 DA FD | JSR \$FDDA |                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 0310- | 60       | RTS        |                                                                                                                                                                                                                                                                                                                                                                                                                                |



It should be pointed out that the decimal mode affects arithmetic that takes place in the accumulator, but does not change the way in which INY or INX works when registers are incremented. The decimal mode does not affect the way in which INC works when regular memory locations are incremented either. Similar statements can be made about the DEX, DEY, and DEC commands. Consider the program:

| <i>Rough Assembly Version</i> | <i>Finished Assembly Listing</i> |
|-------------------------------|----------------------------------|
| SED                           | 0300- F8 SED                     |
| LDA #00                       | 0301- A9 00 LDA #\$00            |
| LDY #00                       | 0303- A0 00 LDY #\$00            |
| CLC                           | 0305- 18 CLC                     |
| ADC #01                       | 0306- 69 01 ADC #\$01            |
| INY                           | 0308- C8 INY                     |
| CPY #10                       | 0309- C0 10 CPY #\$10            |
| BNE ?                         | 030B- D0 F8 BNE \$0305           |
| JSR FDDA                      | 030D- 20 DA FD JSR \$FDDA        |
| RTS                           | 0310- 60 RTS                     |

Y ≠ 10

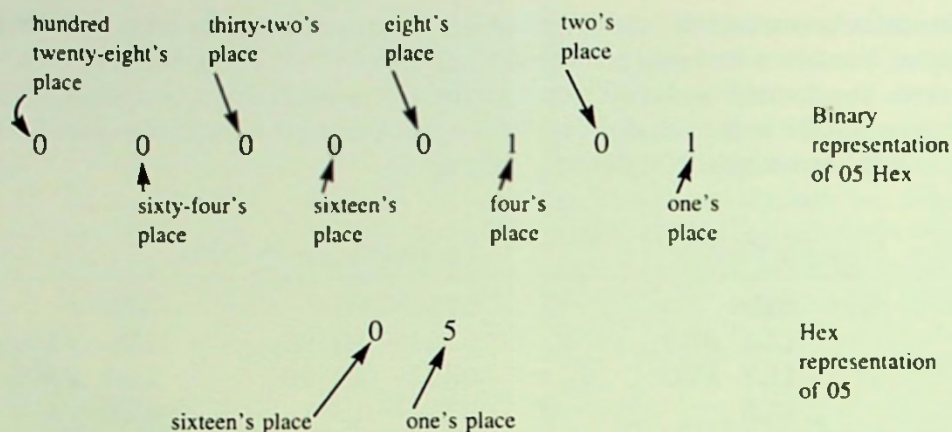
This program would give 16 as its output. The statement ADC #01 adds one to the accumulator each time the loop is executed. The arithmetic in the accumulator is done in decimal. The incrementation in the Y register is done in HEX, however. The upper loop limit \$10 is interpreted to mean "16," not "10," because the operand \$10 is not part of an ADC command. Therefore the loop is executed 16 (\$10) times. The results are displayed as 16 since the accumulator counts the number of times the loop is executed using decimal arithmetic to do the counting.

After a program is run using the decimal mode, the Apple will "beep" its speaker. Usually a beep indicates a syntax error. In decimal mode, however, Apple's beep merely serves as a reminder to the programmer that the decimal mode is set. To clear the decimal mode, use the implied mode command CLD (*C*lear the *D*ecimal mode, op code D8. The monitor Go command automatically clears the decimal mode. CLD does not have to be used to clear the decimal mode in any program not containing SED, even if the program is executed right after another program that was run in decimal mode).

## Reading in Two-Digit Numbers from the Keyboard

We can use the ADC command to help us enter two-digit numbers from the keyboard. Two "Read the Keyboard" commands (JSR FD35) will be used to read each of the two digits separately. The two digits will then be combined to give a single two-digit number. Another command we will use to combine the two digits into a single number is the ASL (Arithmetic Shift Left, op code is 0A) command. Its effect is described next.

Suppose the HEX number 05 is stored in the accumulator. In reality, this number is stored in binary as shown here.



The first 4 bits store the nibble 0 and the last 4 bits store the nibble 5. In binary, the value of each place is two times larger than the place immediately to the right (in HEX, the place value of the left nibble is 16 times the place value of the right nibble).

When ASL is executed, each bit stored in the accumulator is moved one place to the left (see Appendix B for the op codes needed to ASL a regular memory location). There is no bit to fill the space vacated by the old rightmost bit, so the rightmost bit is filled with a zero. The old leftmost bit is moved into the Carry bit. Thus each time ASL is executed, the value that used to be stored in the Carry bit is lost because the left bit in the HEX number being ASLed replaces it. Schematically we have

|            | Carry Bit | Left Nibble | Right Nibble | HEX Representation |
|------------|-----------|-------------|--------------|--------------------|
| Before ASL | 0         | 0 0 0 0     | 0 1 0 1      | 05                 |
| After ASL  | 0         | 0 0 0 0     | 1 0 1 0      | 0A = 05 × 2        |

The effect of ASL is to move each bit into a place whose value is two times the bit's old place value. The net effect is to multiply the number being ASLed by two (unless the leftmost bit that "gets pushed out of the accumulator" into the C bit is a one).

Suppose we execute three more ASL commands:

|            | Carry Bit | Binary Representation | HEX Representation    |
|------------|-----------|-----------------------|-----------------------|
| Second ASL | 0         | 00010100              | 14 = 0A × 2 = 05 × 4  |
| Third ASL  | 0         | 00101000              | 28 = 14 × 2 = 05 × 8  |
| Fourth ASL | 0         | 01010000              | 50 = 28 × 2 = 05 × 10 |



A few comments need to be made here. First, even though the C bit looks as though it is not being changed, each zero shown is a "different zero." It is the zero that used to be in the leftmost bit in the accumulator before ASL was executed. Second, note that each ASL command multiplies the previous value stored in the accumulator by two. The net effect of four ASL's is to multiply the original number by 16 (\$10). In HEX, multiplying by 16 can be done mechanically by moving the digits in a number one place to the left and inserting a zero placeholder on the right. Thus  $\$05 \times \$10 = \$50$ . [We have an analogous situation in base 10 (decimal). If you shift the digits in any number one place to the left, the number is multiplied by 10.]

The previous discussion of ASL showed that multiplying a two-digit HEX number by \$10 "moves the digits one place to the left." We will use this fact to help us combine two digits read from the keyboard (say 3 and 6) into a single two-digit number. We will follow this plan:

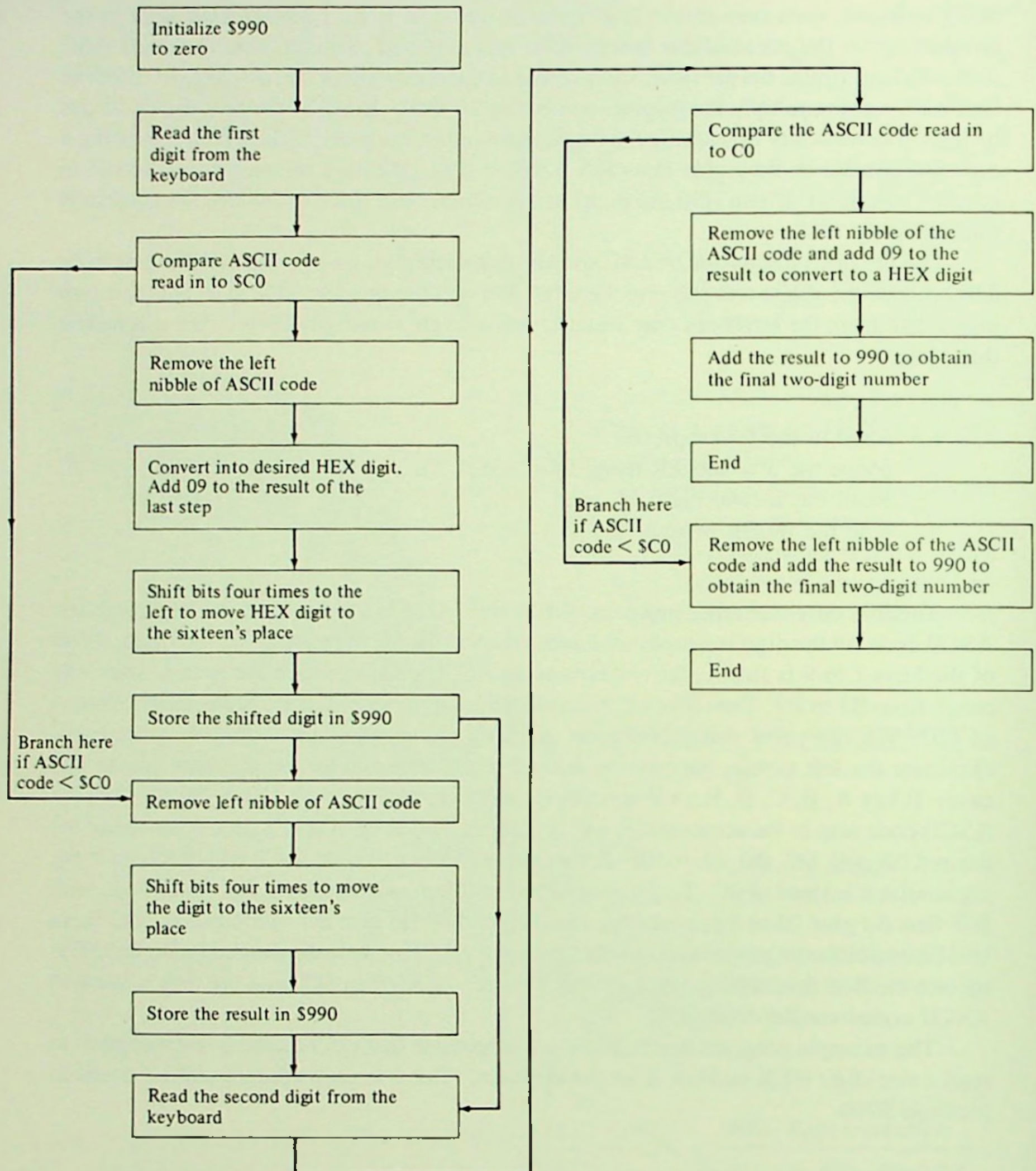
1. Read in the first digit: 03.
2. Move the 3 to the left using four ASLs: 30.
3. Read the second digit: 06.
4. Add the results of steps 2 and 3:  $30 + 06 = 36$ .

There is only one other problem. When JSR FD35 is used to read the keyboard, the ASCII code for the digit is stored in the accumulator, not the digit itself. For example, if one of the keys 1 to 9 is struck, the corresponding ASCII code stored in the accumulator will range from B1 to B9. Thus if key 2 is struck, \$B2 will be stored in the accumulator instead of \$02. We can solve this problem by ANDing the accumulator with \$0F, which will eliminate the left nibble, leaving the desired result stored in the accumulator (02 in this case). If key A, B, C, D, E, or F is struck (remember, these are valid HEX digits), then the ASCII code sent to the accumulator will be either C1, C2, C3, C4, C5, or C6 instead of the desired 0A, 0B, 0C, 0D, 0E, or 0F. For example, if key C is struck, C3 will be stored in the accumulator instead of 0C. To get around this problem, we can AND the ASCII code with \$0F first (to give 03 in this example) and then add 09 (to give the desired 0C). Thus there will be two different procedures for converting an ASCII code to the desired HEX digit. The second method described is used on ASCII codes larger than \$C0 and the first is used on ASCII codes smaller than \$C0.

The example program that follows is a subroutine that can be used in any program to read a two-digit HEX number from the keyboard. The two-digit number will be stored in location \$990.

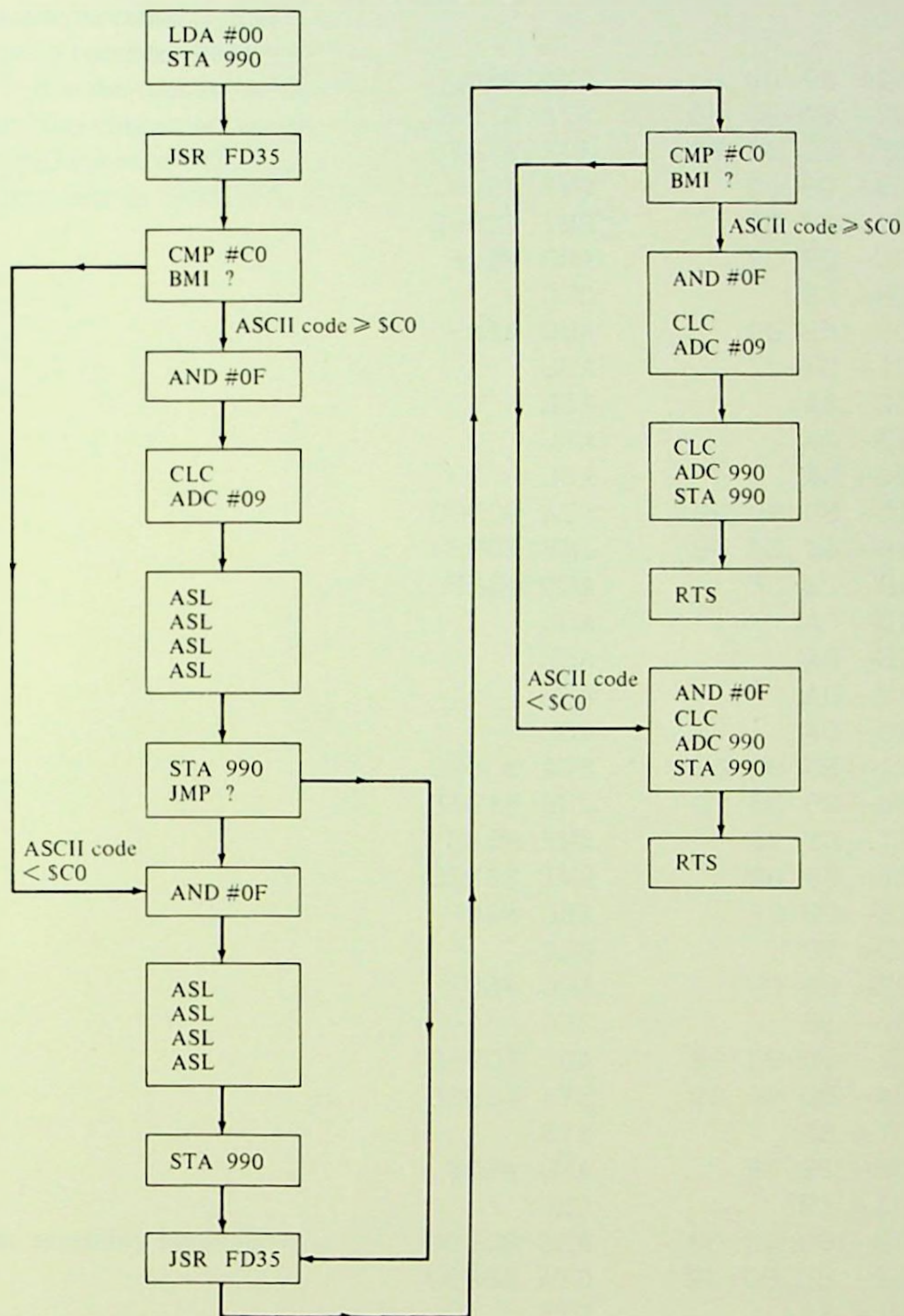


Flowchart1





## Assembly Language Version



The finished assembly listing is shown here:

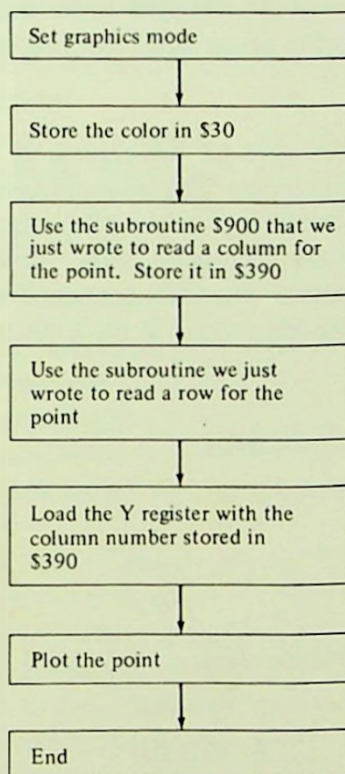
|       |          |            |
|-------|----------|------------|
| 0900- | A9 00    | LDA #\$00  |
| 0902- | 8D 90 09 | STA \$0990 |
| 0905- | 20 35 FD | JSR \$FD35 |
| 0908- | C9 C0    | CMP #\$C0  |
| 090A- | 30 0F    | BMI \$091B |
| 090C- | 29 0F    | AND #\$0F  |
| 090E- | 18       | CLC        |
| 090F- | 69 09    | ADC #\$09  |
| 0911- | 0A       | ASL        |
| 0912- | 0A       | ASL        |
| 0913- | 0A       | ASL        |
| 0914- | 0A       | ASL        |
| 0915- | 8D 90 09 | STA \$0990 |
| 0918- | 4C 24 09 | JMP \$0924 |
| 091B- | 29 0F    | AND #\$0F  |
| 091D- | 0A       | ASL        |
| 091E- | 0A       | ASL        |
| 091F- | 0A       | ASL        |
| 0920- | 0A       | ASL        |
| 0921- | 8D 90 09 | STA \$0990 |
| 0924- | 20 35 FD | JSR \$FD35 |
| 0927- | C9 C0    | CMP #\$C0  |
| 0929- | 30 0D    | BMI \$0938 |
| 092B- | 29 0F    | AND #\$0F  |
| 092D- | 18       | CLC        |
| 092E- | 69 09    | ADC #\$09  |
| 0930- | 18       | CLC        |
| 0931- | 6D 90 09 | ADC \$0990 |
| 0934- | 8D 90 09 | STA \$0990 |
| 0937- | 60       | RTS        |
| 0938- | 29 0F    | AND #\$0F  |
| 093A- | 18       | CLC        |
| 093B- | 6D 90 09 | ADC \$0990 |
| 093E- | 8D 90 09 | STA \$0990 |
| 0941- | 60       | RTS        |



The program was loaded into memory beginning in location \$900 so that it can be used as a subroutine to be called in other machine language programs. As you can see, the subroutine for inputting numbers from the keyboard is rather involved. In most cases, you will want simply to use the monitor to enter data into memory.

The following example uses the subroutine just written to input a row R and a column C from the keyboard. The low-resolution graphics point (C,R) is then plotted. Note how JSR is used just as GOSUB is in BASIC.

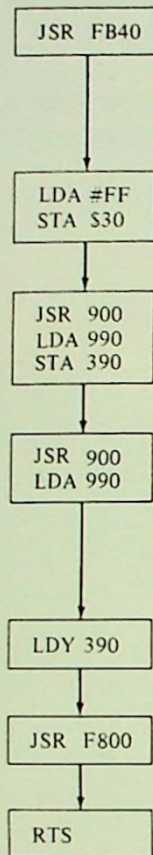
#### Flowchart



The assembly language program corresponding to the flowchart is as follows:

## Assembly Language

## Comments



Set the Graphics mode before loading the color. The subroutine located at \$FB40 can erase any color value stored in \$30.

The RTS command at the end of subroutine \$900 transfers control to the statement following the JSR 900 command.

The value for the column of the point must be stored for safekeeping. The column number was not transferred directly to the Y register since the subroutine \$FD35, which is called in the subroutine \$900, scrambles the contents of the Y register. The RTS command at the end of subroutine \$900 transfers control to the LDA 990 instruction.

Plot the point.

A completed listing is presented here.

|       |    |    |    |            |
|-------|----|----|----|------------|
| 0300- | 20 | 40 | FB | JSR \$FB40 |
| 0303- | A9 | FF |    | LDA #\$FF  |
| 0305- | 85 | 30 |    | STA \$30   |
| 0307- | 20 | 00 | 09 | JSR \$0900 |
| 030A- | AD | 90 | 09 | LDA \$0990 |
| 030D- | 8D | 90 | 03 | STA \$0390 |
| 0310- | 20 | 00 | 09 | JSR \$0900 |
| 0313- | AD | 90 | 09 | LDA \$0990 |
| 0316- | AC | 90 | 03 | LDY \$0390 |
| 0319- | 20 | 00 | F8 | JSR \$F800 |
| 031C- | 60 |    |    | RTS        |



Before moving on, it is worth mentioning that sometimes a perfectly logical machine language program will not work! Consider the following alternate version of the program just considered.

| <i>Assembly Language</i> | <i>Comments</i>                                          |
|--------------------------|----------------------------------------------------------|
| JSR FB40                 | Set Graphics mode.                                       |
| LDA #FF                  |                                                          |
| STA \$30                 | Store the color for the point.                           |
| JSR 900                  | Read a two-digit column for the point from the keyboard. |
| LDA 990                  |                                                          |
| TAY                      | Move the column number into the Y register.              |
| JSR 900                  | Read a row number for the point from the keyboard.       |
| LDA 990                  |                                                          |
| JSR F800                 | Plot the point.                                          |
| RTS                      | End.                                                     |

This program will *not* work. The subroutine located at \$900 calls the subroutine located at \$FD35 to read the keyboard. The subroutine located at \$FD35 scrambles the Y register. Therefore the column number stored in the Y register is lost when the row number is read from the keyboard. For this reason, the column number was put into storage and transferred to the Y register after using the subroutine \$900 to read the row number for the point in the correct version of the program.

It is worthy of note that the *Apple Reference Manual* does not warn you that use of the subroutine \$FD35 scrambles the contents of the Y register.

The lesson to be learned here is that you need to be very careful about calling built-in subroutines if data stored in the X register, Y register, or accumulator need to be used after the built-in subroutine is called. You should also be aware of the fact that certain commands (CMP, for example) affect bits in the processor status register. If you use branch commands such as BCC to test bits in the P register, you need to make sure commands in your program are not affecting these bits in ways you forgot to consider. Suspect the kind of traps mentioned here if your program "just has to work" but does not.

Finally, you can usually write simple programs that test the effect of subroutines on data in storage if you have suspicions that a subroutine may be causing you to lose values stored in memory. For example,

```
LDA #09
TAY
JSR FD35
TYA
JSR FDDA
RTS
```

does *not* give output 09. This proves that the subroutine located at \$FD35 does indeed scramble the contents of the Y register.

## Subtraction

The mechanics of subtraction are very much like the mechanics of addition. You still must work from right to left. Just as the Apple automatically took care of carries in addition, it will automatically handle "borrowing" when we subtract. The Carry bit keeps track of borrowing. The command we use to subtract is SBC (SuBtract with Carry). Like ADC, the SBC command can be used in immediate, absolute, zero-page, and absolute indexed addressing modes. It can also be used in some addressing modes that we do not study until the last chapter.

Before starting subtraction, the Carry bit must be *set* (equal to one). The reason for this has to do with the way the computer subtracts. At this point, the theoretic details of computer subtraction will not be covered because this would not greatly increase your understanding of machine language. You can use the implied mode command SEC (SEt the Carry bit, op code 38) to set the Carry bit before you subtract. The flowchart presented next outlines the method used to subtract two four-digit numbers.

### Memory Use

390  
Most significant byte  
of number you are  
subtracting from.

392  
Most significant byte  
of the number you  
are subtracting.

394  
Most significant  
byte of the answer.

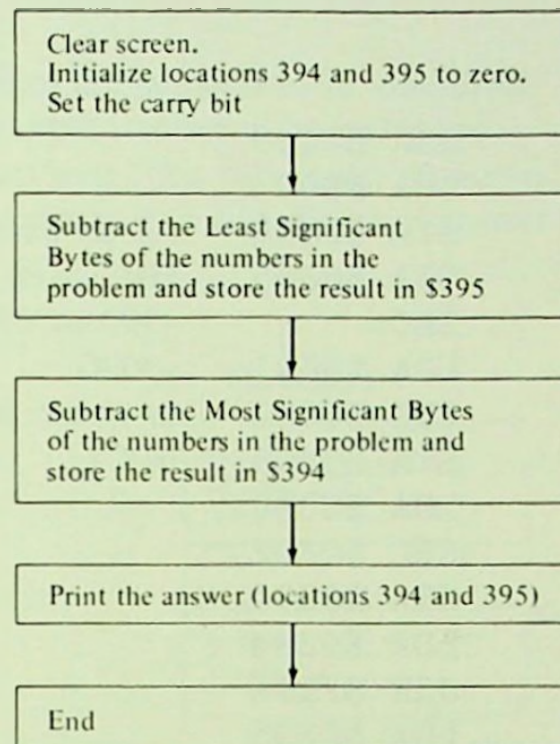
391  
Least significant byte  
of number you are  
subtracting from.

393  
Least significant byte  
of the number you are  
subtracting.

395  
Least significant byte  
of the answer.



## Flowchart



## Assembly Language

## Comments

|                                                   |                                                                                                                                                                                    |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> JSR FC58 LDA #00 STA 394 STA 395 SEC </pre> | <p>The Carry bit is set before any subtraction is done. It is set only at the beginning of the program, not before every pair of bytes is subtracted.</p>                          |
| <pre> LDA 391 SBC 393 STA 395 </pre>              | <p>This program is virtually the same in concept as the program for adding two four-digit numbers. In this program, SEC is used instead of CLC and SBC is used instead of ADC.</p> |
| <pre> LDA 390 SBC 392 STA 394 </pre>              |                                                                                                                                                                                    |
| <pre> LDA 394 JSR FDDA LDA 395 JSR FDDA </pre>    |                                                                                                                                                                                    |
| <pre> RTS </pre>                                  |                                                                                                                                                                                    |

The completed program for subtracting two four-digit numbers would have this listing.

|       |          |            |
|-------|----------|------------|
| 0300- | 20 58 FC | JSR \$FC58 |
| 0303- | A9 00    | LDA #\$00  |
| 0305- | 8D 94 03 | STA \$0394 |
| 0308- | 8D 95 03 | STA \$0395 |
| 030B- | 38       | SEC        |
| 030C- | AD 91 03 | LDA \$0391 |
| 030F- | ED 93 03 | SBC \$0393 |
| 0312- | 8D 95 03 | STA \$0395 |
| 0315- | AD 90 03 | LDA \$0390 |
| 0318- | ED 92 03 | SBC \$0392 |
| 031B- | 8D 94 03 | STA \$0394 |
| 031E- | AD 94 03 | LDA \$0394 |
| 0321- | 20 DA FD | JSR \$FDDA |
| 0324- | AD 95 03 | LDA \$0395 |
| 0327- | 20 DA FD | JSR \$FDDA |
| 032A- | 60       | RTS        |

As an example, you could subtract  
1642 from 8403 using  
\*390:84 03 16 42  
\*300G  
to obtain the output  
6DC1

## Multiplication

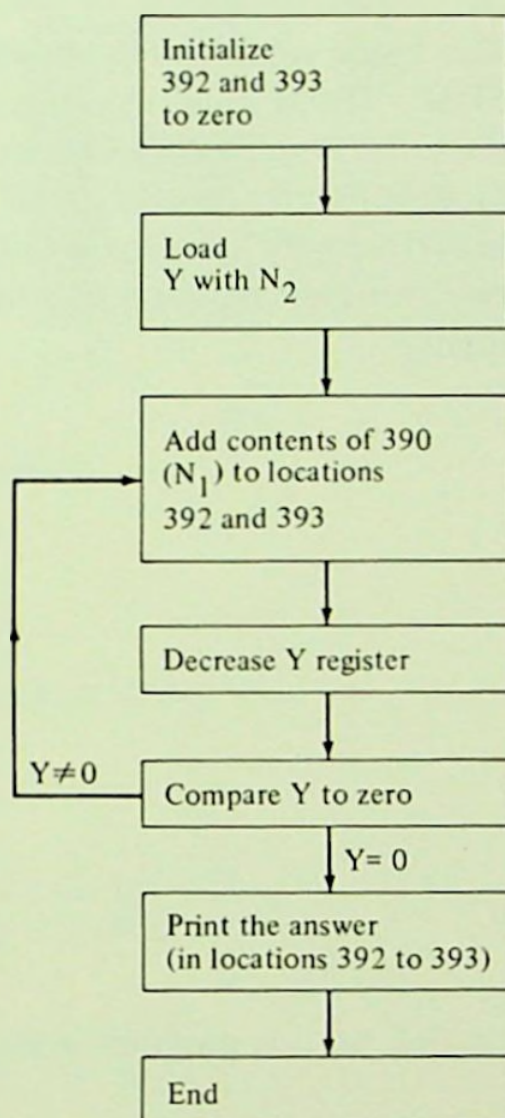
One approach to multiplication is to perform repetitive addition. To multiply two numbers  $N_1$  and  $N_2$ , we can add  $N_1$  to itself  $N_2$  times. A program that accomplishes this is developed in the following:

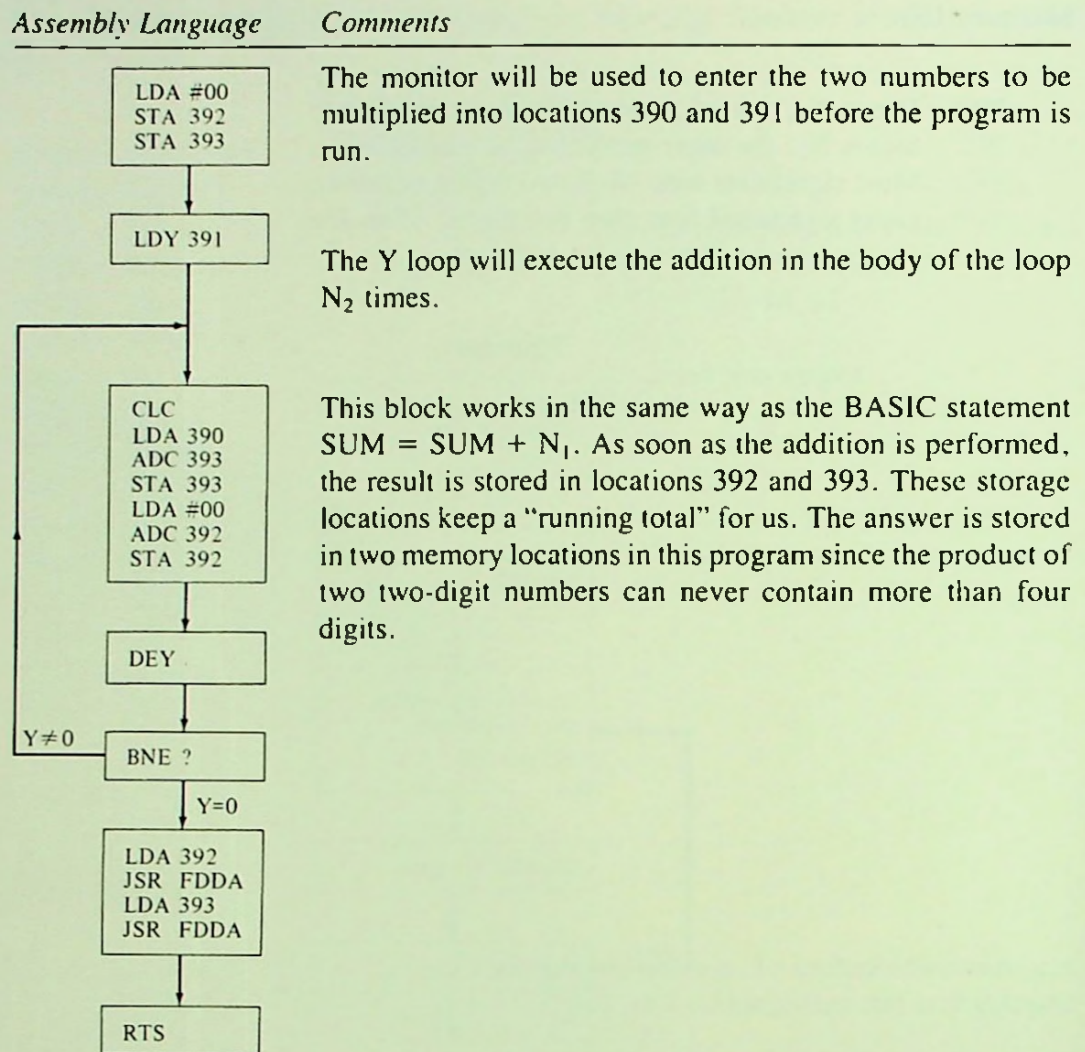


## Memory Use

- 390: Stores  $N_1$ , one of the numbers to be multiplied.  
391: Stores  $N_2$ , the other number to be multiplied.  
392: Most significant byte (first two digits) of answer.  
393: Least significant byte (last two digits) of answer.

## Flowchart





The completed assembly listing for this program is shown here.



```

0300- A9 00      LDA #$00
0302- 8D 92 03   STA $0392
0305- 8D 93 03   STA $0393
0308- AC 91 03   LDY $0391
030B- 18         CLC
030C- AD 90 03   LDA $0390
030F- 6D 93 03   ADC $0393
0312- 8D 93 03   STA $0393
0315- A9 00      LDA #$00
0317- 6D 92 03   ADC $0392
031A- 8D 92 03   STA $0392
031D- 88         DEY
031E- D0 EB      BNE $030B
0320- AD 92 03   LDA $0392
0323- 20 DA FD   JSR $FDDA
0326- AD 93 03   LDA $0393
0329- 20 DA FD   JSR $FDDA
032C- 60         RTS

```

A sample run might be made using

```

* 390: 40 50
* 300G

```

The output would be the product of 40 and 50

1400

We close this chapter by giving an alternative approach to multiplication that is based on the familiar paper-and-pencil multiplication process. Study the following decimal multiplication example.

|                                                              |                                                                                                                                                       |
|--------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\begin{array}{r} 57 \\ \times 36 \\ \hline 342 \end{array}$ | <p>STEP 1. Multiply top number by the right digit in the bottom number to obtain the first row that will be added when the final answer is found.</p> |
|--------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                                                     |                                                                                                                                                                                                                                                                       |
|---------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\begin{array}{r} 57 \\ \times 36 \\ \hline 342 \\ 171 \end{array}$ | <p>STEP 2. Multiply the top number by the left digit in the bottom number to obtain the second row that will be added when the final answer is found. The second row is shifted to the left to take into account the fact that 3 has a higher place value than 6.</p> |
|---------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                                                                    |                                                      |
|------------------------------------------------------------------------------------|------------------------------------------------------|
| $\begin{array}{r} 57 \\ \times 36 \\ \hline 342 \\ 171 \\ \hline 2052 \end{array}$ | <p>STEP 3. Add the rows to get the final answer.</p> |
|------------------------------------------------------------------------------------|------------------------------------------------------|

Binary multiplication is done in much the same way. Let us do the foregoing problem using binary arithmetic. Since

$$57_{10} = \$39 = 00111001_2$$

and

$$36_{10} = \$24 = 00100100_2$$

we need to multiply 00111001 by 00100100.

$$\begin{array}{r}
 00111001 \\
 \times 00100100 \\
 \hline
 00000000 \\
 00000000 \\
 00111001 \\
 00000000 \\
 00000000 \\
 00111001 \\
 00000000 \\
 00000000 \\
 \hline
 00010000000100 \\
 \hline
 0 \quad 8 \quad 0 \quad 4
 \end{array}$$

Row 1 of the work is found by multiplying the top number by the last digit in the bottom number.

We then work with the other digits in the bottom number, working from right to left. Each row of the work is shifted one more place to the left than the row above it. The final answer is found by adding all eight rows of the work.

Answer in binary

Answer in HEX

$$\begin{aligned}
 &0 \times 16^3 + 8 \times 16^2 + 0 \times 16 + 4 \\
 &= 8 \times 256 + 4 = 2052, \text{ as before}
 \end{aligned}$$

Answer in decimal

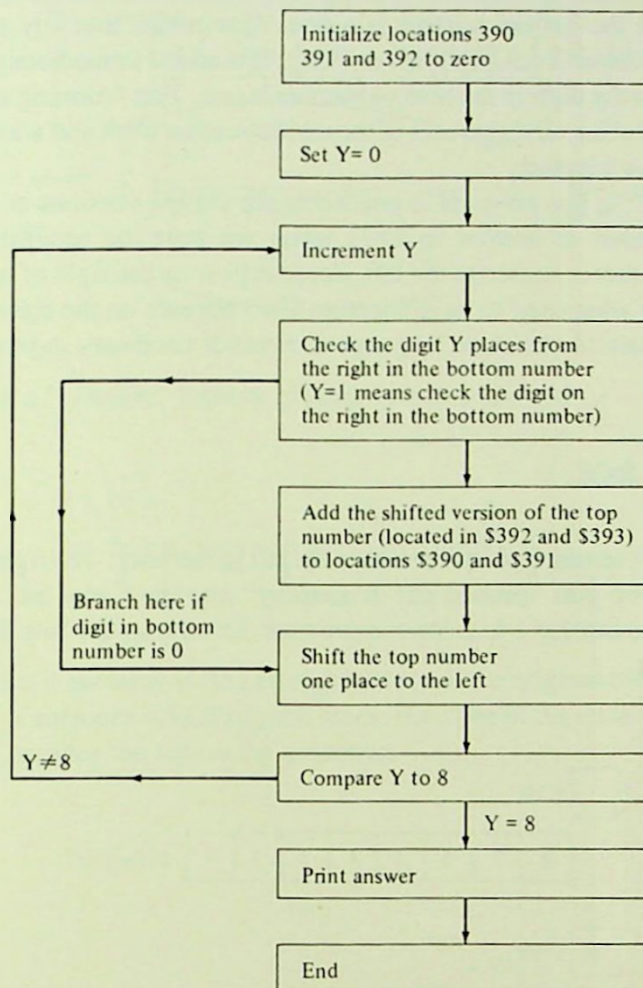
Examine the binary multiplication problem carefully. You should see that only two rows of the work are important, namely, the third row and the sixth row. Each row of the work consists of either all zeros (if the corresponding digit in the bottom number is a zero) or else is a left-shifted copy of the top number (if the corresponding digit in the bottom number is a one). Using these observations, we can construct this multiplication flowchart:

### Memory Use

- 390 and 391 will contain the most significant and least significant bytes of the answer.
- 392 and 393 contain the left-shifted copy of the top number in the multiplication problem. You enter the top number to be multiplied into location \$393 and the bottom number to be multiplied into location \$394 using the monitor.



## Flowchart



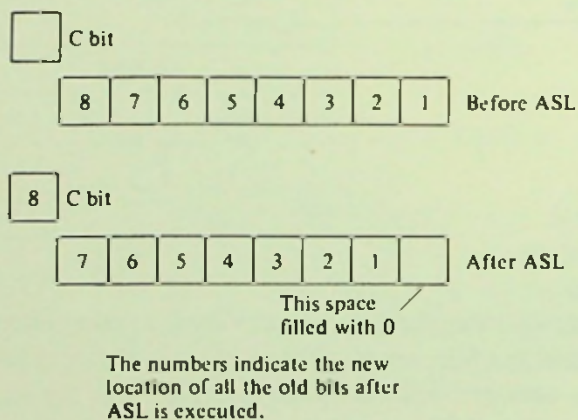
Each time through the loop, the digit being examined in the bottom number determines whether the shifted version of the top number is added to the locations that ultimately will contain the complete answer (locations \$390 and \$391). In the example problem, you should see that only two numbers are added to give the final product: 0000000011100100 (the contribution from the third row) and 0000011100100000 (the contribution from the sixth row). For simplicity, the leading and trailing zeros were omitted in the paper-and-pencil work. It is important to realize that, to model the shifting process in the computer, the leading and trailing zeros must be included. Each row of the work can be represented by storing a four-digit HEX number in two memory locations, since each row of the multiplication work is really a 16-digit binary number when leading and trailing zeros are included. Notice that each time through the loop, the top number being multiplied is shifted even if the digit checked in the bottom number is a zero. Using this strategy, the top number is always

shifted the proper number of times and is ready to be added to locations \$390 and \$391, if the digit checked in the bottom number is a one. Also notice that any particular shifted version of the top number is not kept for very long. It is added immediately to the locations \$390 and \$391 when the digit in the bottom number is one. This "running total" approach is much simpler than storing all eight rows of the multiplication work and waiting until the end to add all eight rows together.

The real "trick" to this program is producing the shifted versions of the top number. The entire top number is located in \$393 when we start the program. As the work progresses, this number is shifted to the left. Zeros appear on the right of location \$393 and the leftmost digit of what used to be in location \$393 appears on the right side of location \$392. We next explain the machine language commands necessary to produce this effect.

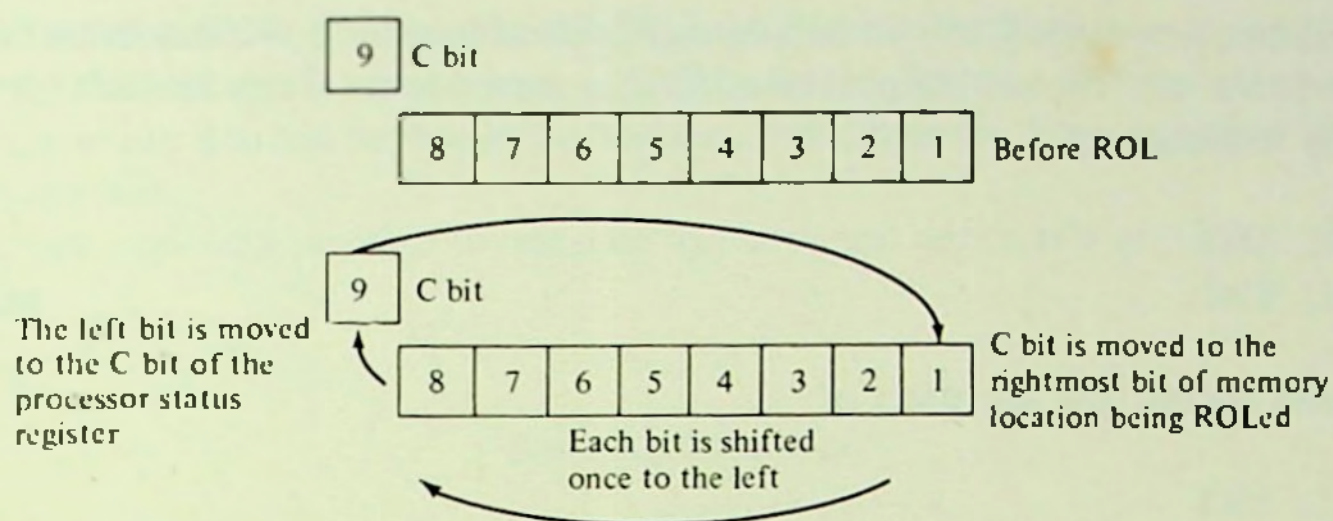
### Use of ASL and ROL

Recall that the ASL command shifts the bits to the left in memory. The right bit is filled with a zero and the left bit gets "pushed out of memory" into the Carry bit. In the following diagram, the bits are labeled 1 to 8 for convenience. Of course, the bits themselves are all either ones or zeros.

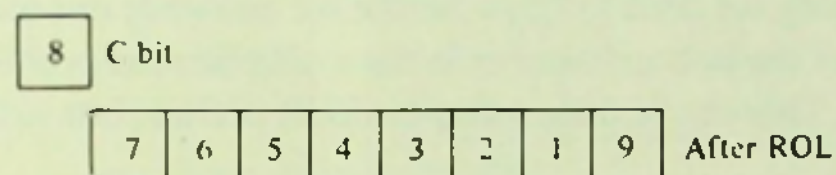


ROL (*RO*tate bits to the *L*eft) can be used in many different addressing modes. (See Appendix B for the various op codes of this command.) The ROL command is similar to the ASL command except that the bit on the right is not filled with a zero as when ASL is used. It is filled with the old contents of the Carry bit as shown in this diagram.





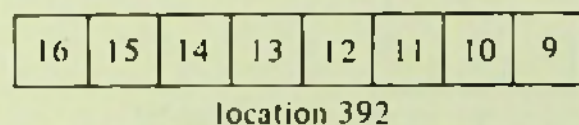
ROL acts in a “circular” fashion moving bits as indicated.



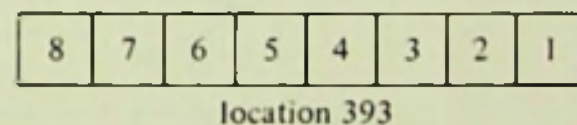
You should see that if memory is ASLed eight times, all the original bits will be replaced by zeros. However, if memory is ROLed nine times, it will be in the same state it was in before the first ROL. Consider the following sequence:

ASL 393  
ROL 392

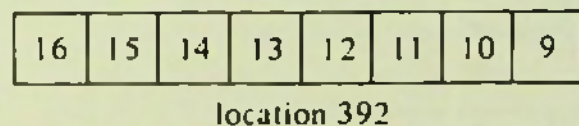
Before execution



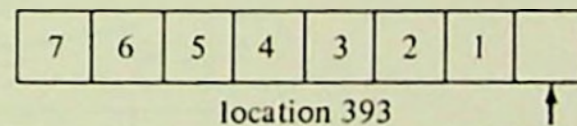
C bit



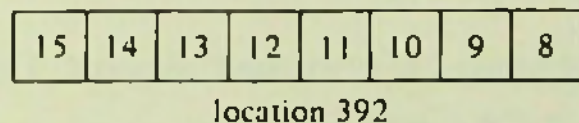
After execution of ASL 393



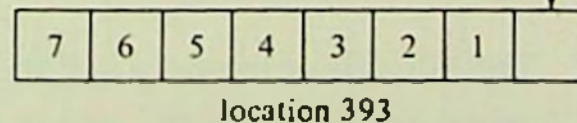
8 C bit



After execution of ASL 393  
ROL 392



16 C bit



Filled with zero

ASL 393 shifts location \$393 once to the left and sends its leftmost bit to the Carry bit in the processor status register. The rightmost bit of \$393 is filled with zero. The ROL 392

command then moves the Carry bit into the right side of location \$392 and shifts the other bits once to the left. The old leftmost bit of \$392 is moved to the Carry bit (this bit will be lost if the sequence

```
ASL 393
ROL 392
```

is executed again). The net effect of

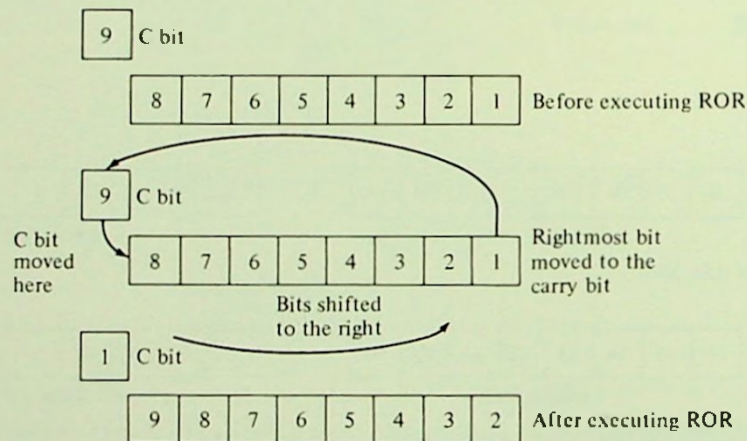
```
ASL 393
ROL 392
```

is to link the locations \$392 and \$393 together so that what gets "pushed out of the left side of \$393 appears on the right side of \$392."

The only other thing we need to know before we can write our multiplication program is how to test the bits of the bottom number in the multiplication problem, starting from the right and working left. This can be done using the ROR (ROtate bits to the Right) command.

## ROR Command

The ROR command works just as the ROL command does, except that the bits are moved to the right instead of to the left.



The left side of memory is filled with what used to be in the Carry bit. The rightmost bit is "pushed out of memory" and is placed in the Carry bit.

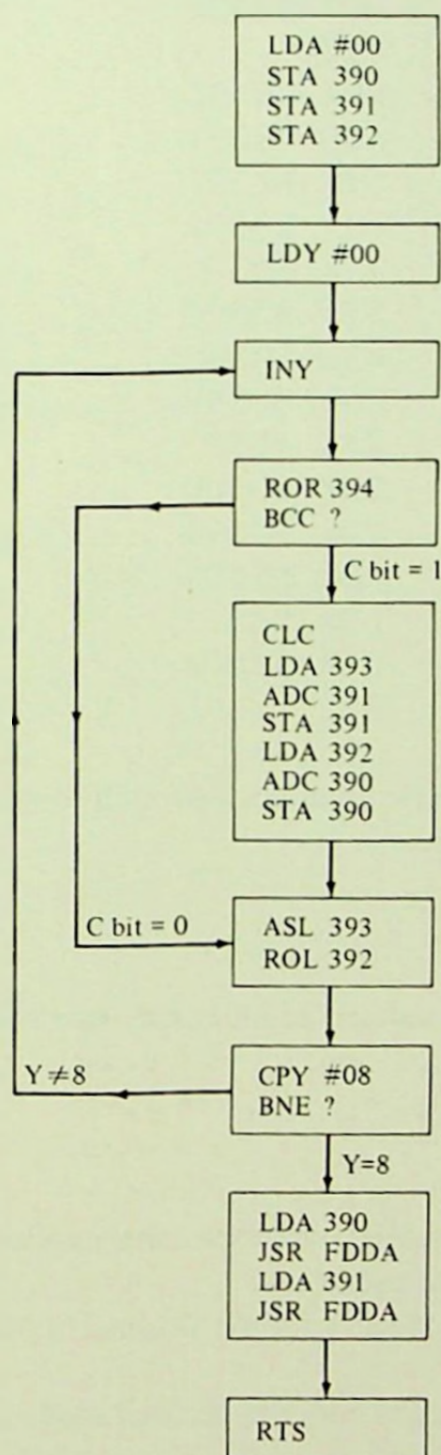
We can use the ROR command together with a BCC command to test the bits of the bottom number in the multiplication problem. Suppose the bottom number in the problem is



stored in location \$394. The command ROR 394 will move the first bit on the right into the Carry bit, where it can be tested with a BCC command. Because ROR rotates bits to the right, we can use it to test the bits in the bottom number in order, from rightmost bit first to leftmost bit last.

We are now in a position to translate the flowchart previously given into assembly language.

### Assembly Language



The completed assembly language listing for the program is

```

0300- A9 00      LDA #$00
0302- 8D 90 03   STA $0390
0305- 8D 91 03   STA $0391
0308- 8D 92 03   STA $0392
030B- A0 00      LDY #$00
030D- C8         INY
030E- 6E 94 03   ROR $0394
0311- 90 13      BCC $0326
0313- 18         CLC
0314- AD 93 03   LDA $0393
0317- 6D 91 03   ADC $0391
031A- 8D 91 03   STA $0391
031D- AD 92 03   LDA $0392
0320- 6D 90 03   ADC $0390
0323- 8D 90 03   STA $0390
0326- 0E 93 03   ASL $0393
0329- 2E 92 03   ROL $0392
032C- C0 08      CPY #$08
032E- D0 DD      BNE $030D
0330- AD 90 03   LDA $0390
0333- 20 DA FD   JSR $FDDA
0336- AD 91 03   LDA $0391
0339- 20 DA FD   JSR $FDDA
033C- 60         RTS

```

To run the program, store the first number you wish to multiply in \$393 and the second number in \$394, and then type

\*300G

The program can be used to multiply together any two two-digit numbers.

### Review Questions

1. Before adding two HEX numbers, the implied addressing mode command \_\_\_\_\_ should be used.
2. If 09 and 12 are added with the decimal mode set, then the result will be \_\_\_\_\_.
3. If 09 and 12 are added with the decimal mode cleared, then the result will be \_\_\_\_\_.
4. The number \$39 is stored in the accumulator. If the command ASL is executed, then \$\_\_\_\_\_ will be stored in the accumulator.



5. Before using the subtraction command SBC, the implied addressing mode command \_\_\_\_\_ should be used.
6. Perform the multiplication in binary and write your answer in HEX:

$$\begin{array}{r} 01101110 \\ \times 10011001 \\ \hline \end{array}$$

7. The HEX number \$82 is stored in location \$350 and the HEX number \$02 is stored in location \$351. After executing the commands

```
ASL $350
ROL $351
```

\$\_\_\_\_\_ will be stored in location \$350 and \$\_\_\_\_\_ will be stored in \$351. What value will be stored in the Carry bit of the processor status register?

\_\_\_\_\_

8. The HEX number \$4D is stored in location \$350. After executing ROR \$350, what value will be stored in \$350? \$\_\_\_\_\_ What value will be stored in the Carry bit of the processor status register? \_\_\_\_\_ (The Carry bit was clear before executing ROR \$350.)

#### TRUE or FALSE

9. There is no machine language command that will multiply two HEX numbers directly. To multiply two numbers in machine language, one needs to write a machine language program.
10. The subtraction command SBC automatically takes care of "borrowing" for you.
11. If two 2-byte numbers are added in a machine language program, then the command CLC should be used twice in the program.
12. If a program is run with the decimal mode set, then Apple will beep its speaker after the program is executed, even if there are no syntax errors in the program.
13. The decimal mode affects the way in which the command INX operates.
14. If a number less than \$80 is stored in the accumulator, then executing ASL multiplies the accumulator's contents by two.
15. The output of the program segment

```
LDY #08
JSR FD35
STA 390
TYA
JSR FDDA
```

is 08.

**Programming Problems**

1. Write a machine language program that prints the binary representation for any two-digit number entered from the keyboard. Use the ASL and BCC commands in your program.
2. In the song "The Twelve Days of Christmas," the singer receives
  - 1 gift on the first day
  - 1 + 2 gifts on the second day
  - 1 + 2 + 3 gifts on the third day
  - 1 + 2 + 3 + 4 gifts on the fourth day

1 + 2 + 3 + ... + 12 gifts on the last day

Write a program to compute (in HEX) the total number of gifts received on all 12 days.

3. Write a machine language program to print out a HEX multiplication table. Since 16 entries will not fit on the screen horizontally, you may want to limit your table to the A's times table:

```

      01 02 03 04 ... 09 0A
01
02  The entry in row R, column C
03  should be R*C, of course.
.
.
.
0A
```

4. Write a machine language program that divides by successive subtraction. Store a four-digit HEX number in two memory locations. Divide this number by a two-digit HEX number stored in another location. Print the quotient (whole-number answer) and any remainder that result from the division.
5. Write a machine language program that allows the user to input a two-digit number from the keyboard and then prints this number with its digits reversed.



# Chapter 7

## Shape Tables

To manipulate high-resolution graphics shapes effectively (enlarge them, rotate them, or move them around on Apple's screen), you use *shape tables*. The topic of shape tables is not strictly entirely in the machine language domain, since BASIC functions are used to convert the codes in a shape table into the high-resolution shapes that appear on the screen. The codes that make up the shape table consist of binary bits, so that a knowledge of machine language is helpful, however. There are several steps that must be carried out to construct a shape table.

1. Construct a *shape definition*.
  - Draw the shape on a grid. Determine the *plotting vectors* required to trace out the shape.
  - Place the binary code for each plotting vector into the bytes that make up the shape definition.
  - Convert the binary codes in each byte into two-digit HEX codes.
2. Construct an *index*.

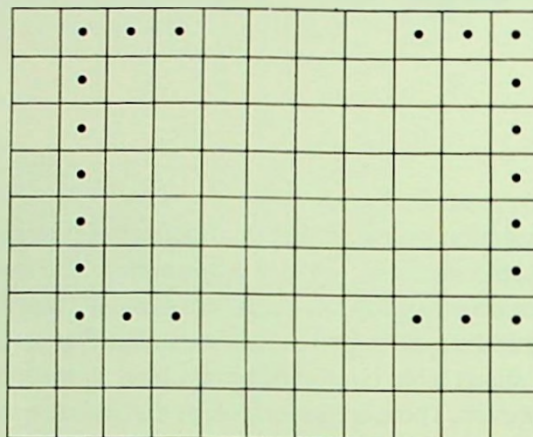
The index will contain data telling Apple how many shape definitions are in the shape table and the location of each shape definition in the table relative to the first byte in the table.

### Plotting Vectors

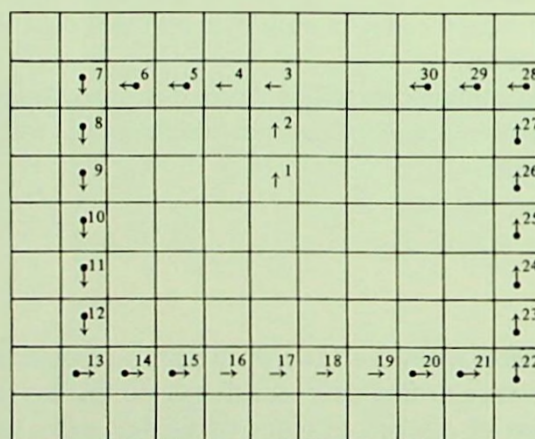
As explained, a shape table consists of two main parts—an *index* and a *shape definition*. The index tells Apple where to find the codes that determine the shape to be drawn. The shape definition consists of a series of bytes containing codes for plotting vectors. To determine these plotting vectors, you first need to plot your shape on a grid. Once each dot of the high-resolution shape has been plotted on your grid, pick a starting point and trace a path through each point in the shape. In tracing the path, you are allowed only to move up, down, right, or left. Each move you make in tracing out the shape is represented by one of the plotting vectors. For example, ↑ means you took one step up, and ← represents one

step to the left. You can move without plotting a point in order to get to the next square containing a point that needs to be plotted. Thus some vectors will represent "plot a point and then move" whereas others will just move you to the next square that has a point in it. The notation  $\uparrow$  will mean you took one step up without plotting a point, and the notation  $\uparrow\bullet$  will mean a high-resolution point was plotted before taking one step up.

Suppose we wish to plot this shape:  The grid shows the points that need to be plotted to produce the shape.



The starting point is arbitrary. Therefore there are many ways to trace out the same shape. It is usually convenient to choose a starting point near the geometric center of the shape, however.



The arrows represent the path you took in tracing out your shape. Apple will draw your shape by retracing your steps. Thus the arrows are used to indicate the path Apple will take



when it draws your shape. If we list the arrows in the order in which they are used, we obtain the following.

| Square number                 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|-------------------------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Corresponding plotting vector | ↑ | ↑ | ← | ← | ↖ | ↖ | ↑ | ↑ | ↑ | ↑  | ↑  | ↑  | ↗  | ↗  | ↗  | →  | →  | →  | →  | →  | ↗  | ↗  | ↑  | ↑  | ↑  | ↑  | ↑  | ↖  | ↖  | ↖  |

As you can see, this work is rather tedious, because each square you pass through in tracing out the high-resolution shape needs to be represented by a plotting arrow.

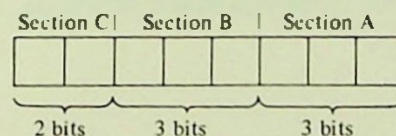
### Binary Codes for Plotting Arrows and Placement of the Binary Codes into the Byte of a Shape Definition

Each plotting arrow has a two- or three-digit binary code:

|             |   |                               |
|-------------|---|-------------------------------|
| ↑ 000       | } | Move without plotting a point |
| → 001 or 01 |   |                               |
| ↓ 010 or 10 |   |                               |
| ← 011 or 11 |   |                               |
| ↑ 100       | } | Plot a move and then move     |
| ↗ 101       |   |                               |
| ↑ 110       |   |                               |
| ↖ 111       |   |                               |

The codes for each plotting vector have to be stored in the bytes that make up the shape definition. We will think of each byte in the shape definition as being divided into three compartments or sections, into which we can place the codes for the plotting arrows. The compartments are filled from right to left (low bit to high bit):

Diagram of a byte in a shape definition



You can see that each byte of the shape definition can hold codes for up to three plotting vectors. There are restrictions on the placement of codes in the shape definition bytes:

1. Section C is smaller than section A or B. Therefore only vectors with two-digit

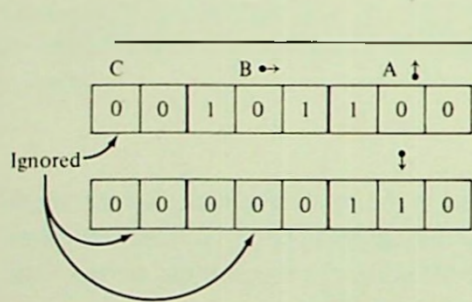
codes ( $\rightarrow$   $\downarrow$   $\leftarrow$ ) can be represented by a binary code in section C. This means that the last code in each byte can never cause a point to be plotted.

2. No section can contain a code of 000 (move up without plotting) unless a section further to the left contains a nonzero code. If a section and all the remaining sections to the left contain zero codes, then all these sections will be ignored. Notice that this restriction means that zeros in section C are always ignored. Therefore the code 00 in section C cannot be used to represent the plotting vector  $\uparrow$ . Furthermore, if any byte contains three sections with zeros, then that byte will mark the end of the shape definition. The next byte with nonzero codes will be assumed to belong to the shape definition of another shape (a shape table can contain shape definitions for up to 255 shapes). Note that the fact that no byte in a shape definition can have three sections all equal to zero without terminating the shape definition means that a shape definition can never contain three consecutive "move up without plotting a point" vectors.

The following examples will illustrate the placement of the binary codes for plotting vectors into the bytes of a shape definition.

### Example 1:

Place the codes for  $\downarrow$   $\rightarrow$   $\uparrow$  into the bytes of a shape definition.



### Comments:

The  $\uparrow$  code will not fit into section C of the first byte and so must be placed in section A of the second byte. Section C of the first byte and sections B and C of the second byte were never filled. They contain zeros and are ignored since no nonzero codes exist to the left of these sections. Notice that the bytes of the shape definition are filled working from right to left.

### Example 2:

Place the codes  $\downarrow$   $\uparrow$   $\rightarrow$   $\uparrow$  into the bytes of a shape definition.



|        | C |   |   | B |   |   | A ↓ |   | Comments                   |
|--------|---|---|---|---|---|---|-----|---|----------------------------|
| Byte 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0   | 0 | Correct placement of codes |
| Byte 2 | 0 | 0 | 1 | 0 | 1 | 0 | 0   | 0 |                            |
| Byte 3 | 0 | 0 | 0 | 0 | 0 | 1 | 1   | 0 |                            |

Notice that

|  | C |   |   | B ↑ |   |   | A ↓ |   | Comments                     |
|--|---|---|---|-----|---|---|-----|---|------------------------------|
|  | 0 | 0 | 0 | 0   | 0 | 1 | 0   | 0 | Incorrect placement of codes |
|  | 0 | 0 | 1 | 1   | 0 | 1 | 0   | 1 |                              |

would be incorrect since there are no nonzero codes to the left of section B in the first byte. Therefore the 000 in section B would be ignored. Apple would interpret the codes in the incorrect shape definition as giving the plotting instructions ↓ → ↓ instead of ↓ ↑ → ↓.

### Example 3:

Place the plotting codes for ↑ ↓ → → ↓ into the bytes of a shape definition.

|  | C → |   |   | B ↓ |   |   | A ↑ |   | Comments                                                                                                                                                                                                                                                       |
|--|-----|---|---|-----|---|---|-----|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | 0   | 1 | 1 | 0   | 0 | 0 | 0   | 0 | The plotting vector → can be represented by the two-bit code 01, which allows it to be stored in section C of the first byte. The → vector is represented by the code 001 if it is to be stored in section A or B, however, as illustrated in the second byte. |
|  | 0   | 0 | 1 | 1   | 0 | 0 | 0   | 1 |                                                                                                                                                                                                                                                                |

**Example 4:**

Place the plotting codes for ↓ ↓ ↔ ↔ ↑ ↓ into the bytes of a shape definition.

*Comments*

| C |   |   | B ↓ |   |   | A ↓ |   |
|---|---|---|-----|---|---|-----|---|
| 0 | 0 | 0 | 1   | 0 | 0 | 1   | 0 |
| ↔ |   |   |     |   |   |     |   |
| 0 | 0 | 1 | 1   | 1 | 1 | 1   | 1 |
| ↓ |   |   |     |   |   |     |   |
| 0 | 0 | 1 | 0   | 0 | 0 | 0   | 0 |

Only two plotting arrows can be stored in the first byte since ↔ has a 3-bit code and will not fit into section C. The ↑ arrow cannot be stored in section C of the second byte (zeros in section C are ignored) and so its code must appear in section A of the last byte.

**Calculation of the HEX Numbers Making Up the Shape Definition**

Each byte in the shape definition contains 8 bits. Group the bits by fours and use this table to convert each group into a HEX digit. The result will be a two-digit HEX number for each byte filled with plotting codes. These HEX numbers make up the completed shape definition.

| <i>Binary</i> | <i>HEX</i> | <i>Binary</i> | <i>HEX</i> |
|---------------|------------|---------------|------------|
| 0000          | 0          | 1000          | 8          |
| 0001          | 1          | 1001          | 9          |
| 0010          | 2          | 1010          | A          |
| 0011          | 3          | 1011          | B          |
| 0100          | 4          | 1100          | C          |
| 0101          | 5          | 1101          | D          |
| 0110          | 6          | 1110          | E          |
| 0111          | 7          | 1111          | F          |

To illustrate the use of this table, the HEX codes for the shape definitions in each of the four examples of the last section are calculated.



| <i>Example</i> |                                                                                                                                                                   | <i>Bits Grouped by<br/>Section A, B, C</i> | <i>Bits Grouped<br/>by Fours</i> | <i>HEX<br/>Repre-<br/>sentation</i> |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|----------------------------------|-------------------------------------|
| 1              | <div> <div>C</div> <div>B</div> <div>A</div> <div> <div>0</div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div><div>0</div><div>0</div> </div> </div> | 00 101 100                                 | 0010 1100                        | 2C                                  |
|                | <div> <div>0</div><div>0</div><div>0</div><div>0</div><div>0</div><div>1</div><div>1</div><div>0</div> </div>                                                     | 00 000 110                                 | 0000 0110                        | 06                                  |
| 2              | <div> <div>0</div><div>0</div><div>0</div><div>0</div><div>0</div><div>1</div><div>0</div><div>0</div> </div>                                                     | 00 000 100                                 | 0000 0100                        | 04                                  |
|                | <div> <div>0</div><div>0</div><div>1</div><div>0</div><div>1</div><div>0</div><div>0</div><div>0</div> </div>                                                     | 00 101 000                                 | 0010 1000                        | 28                                  |
|                | <div> <div>0</div><div>0</div><div>0</div><div>0</div><div>0</div><div>1</div><div>1</div><div>0</div> </div>                                                     | 00 000 110                                 | 0000 0110                        | 06                                  |
| 3              | <div> <div>0</div><div>1</div><div>1</div><div>0</div><div>0</div><div>0</div><div>0</div><div>0</div> </div>                                                     | 01 100 000                                 | 0110 0000                        | 60                                  |
|                | <div> <div>0</div><div>0</div><div>1</div><div>1</div><div>0</div><div>0</div><div>0</div><div>1</div> </div>                                                     | 00 110 001                                 | 0011 0001                        | 31                                  |
| 4              | <div> <div>0</div><div>0</div><div>0</div><div>1</div><div>0</div><div>0</div><div>1</div><div>0</div> </div>                                                     | 00 010 010                                 | 0001 0010                        | 12                                  |
|                | <div> <div>0</div><div>0</div><div>1</div><div>1</div><div>1</div><div>1</div><div>1</div><div>1</div> </div>                                                     | 00 111 111                                 | 0011 1111                        | 3F                                  |
|                | <div> <div>0</div><div>0</div><div>1</div><div>0</div><div>0</div><div>0</div><div>0</div><div>0</div> </div>                                                     | 00 100 000                                 | 0010 0000                        | 20                                  |



**Example 5:**

Place the plotting codes for the vectors representing the [ ] shape plotted at the beginning of the chapter into the bytes of a shape definition and convert the plotting codes to their HEX equivalents.

|                                                                               | <i>Bits Grouped by<br/>Sections A, B, C.</i> | <i>Bits Grouped<br/>by Fours</i> | <i>HEX Repre-<br/>sentation</i> |
|-------------------------------------------------------------------------------|----------------------------------------------|----------------------------------|---------------------------------|
| <div> <div>C ←</div> <div>B ↑</div> <div>A ↑</div> <div>11000000</div> </div> | 11 000 000                                   | 1100 0000                        | C0                              |
| <div> <div>↔</div> <div>↔</div> <div>00111011</div> </div>                    | 00 111 011                                   | 0011 1011                        | 3B                              |
| <div> <div>↓</div> <div>↔</div> <div>00110111</div> </div>                    | 00 110 111                                   | 0011 0111                        | 37                              |
| <div> <div>↓</div> <div>↓</div> <div>00110110</div> </div>                    | 00 110 110                                   | 0011 0110                        | 36                              |
| <div> <div>↓</div> <div>↓</div> <div>00110110</div> </div>                    | 00 110 110                                   | 0011 0110                        | 36                              |
| <div> <div>↔</div> <div>↓</div> <div>00101110</div> </div>                    | 00 101 110                                   | 0010 1110                        | 2E                              |
| <div> <div>↔</div> <div>↔</div> <div>01101101</div> </div>                    | 01 101 101                                   | 0110 1101                        | 6D                              |
| <div> <div>↔</div> <div>↔</div> <div>01001001</div> </div>                    | 01 001 001                                   | 0100 1001                        | 49                              |
| <div> <div>↔</div> <div>↔</div> <div>00101101</div> </div>                    | 00 101 101                                   | 0010 1101                        | 2D                              |
| <div> <div>↓</div> <div>↓</div> <div>00100100</div> </div>                    | 00 100 100                                   | 0010 0100                        | 24                              |
| <div> <div>↓</div> <div>↓</div> <div>00100100</div> </div>                    | 00 100 100                                   | 0010 0100                        | 24                              |
| <div> <div>↓</div> <div>↓</div> <div>00100100</div> </div>                    | 00 100 100                                   | 0010 0100                        | 24                              |
| <div> <div>↔</div> <div>↔</div> <div>00111111</div> </div>                    | 00 111 111                                   | 0011 1111                        | 3F                              |
| <div> <div>↔</div> <div>↔</div> <div>00000111</div> </div>                    | 00 000 111                                   | 0000 0111                        | 07                              |
| <div> <div>↑</div> <div>↑</div> <div>↑</div> <div>00000000</div> </div>       | 00 000 000                                   | 0000 0000                        | 00                              |

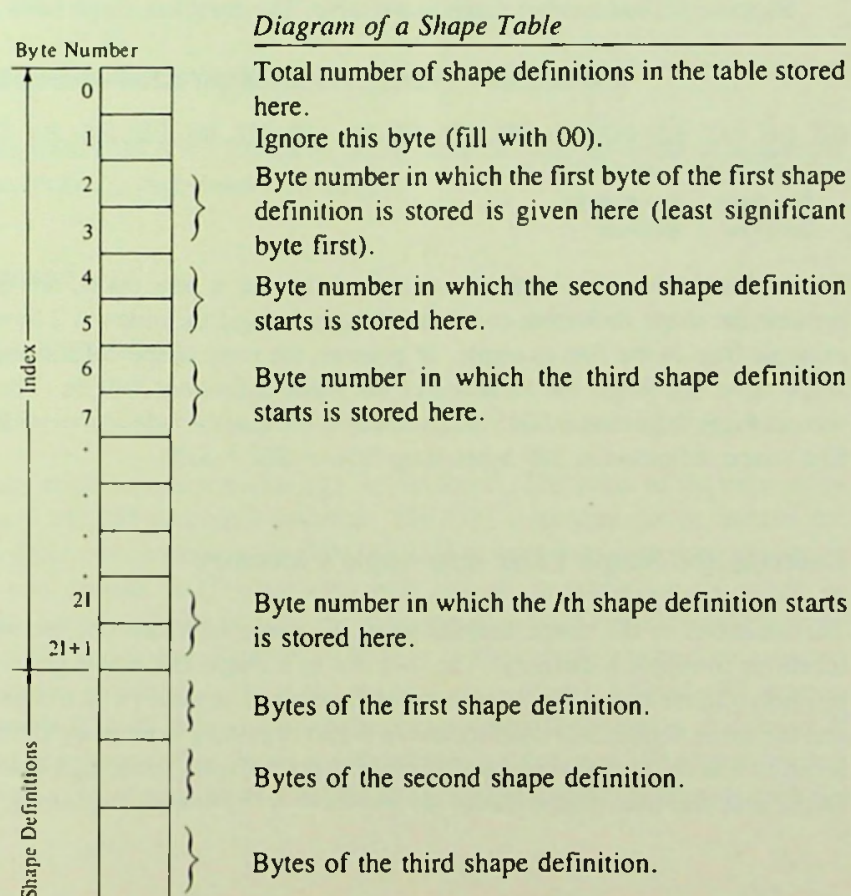


The last byte is needed to signal the end of the shape. It does not cause 3 upward moves. The shape definition for the [ ] shape is thus

C0 3B 37 36 36 2E 6D 49 2D 24 24 24 3F 07 00

### The Index

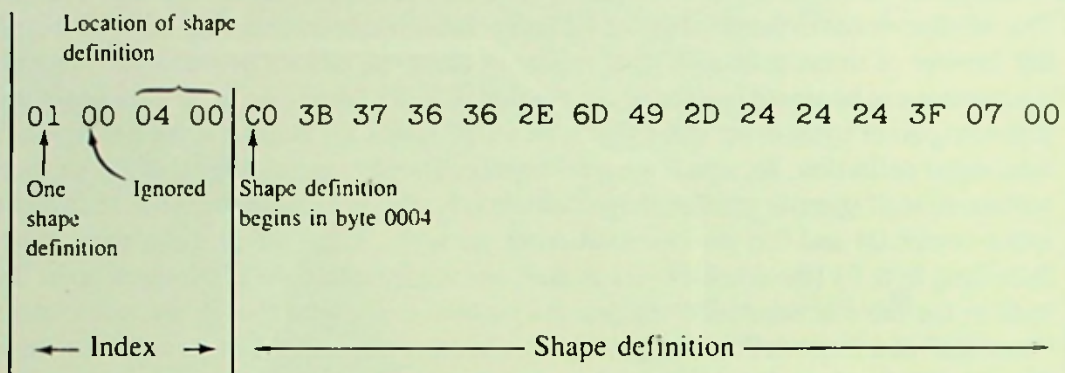
The index of the shape table comes before the HEX codes that make up the shape definition. The number stored in the first byte of the index (labeled byte 0 of the shape table) is equal to the number of shape definitions that appear in the shape table. The maximum number of shapes that can be stored in a shape table is thus \$FF. The next byte in the table is not used. The next pair of bytes in the index (bytes 02 and 03) gives the location of the first byte of the first shape definition. As usual, the least significant byte (last two digits) of the location is written first. If there is another shape definition in the shape table, the next 2 bytes in the index (bytes 04 and 05) give its location in the table. There are as many pairs of bytes following byte 01 (the unused byte) as there are shape definitions in the shape table. Each byte in the table is numbered beginning with byte 0 (the first byte in the index) and the "location" of a shape definition is just the byte number in which the shape definition begins. The location is a four-digit HEX number. A diagram will serve to summarize and clarify the discussion.



It should be emphasized that the first byte in the shape table is labeled zero. Therefore, if your shape definition starts in the fifth byte of the table, its "location" is 0004, since byte 0004 is the fifth byte of the table.

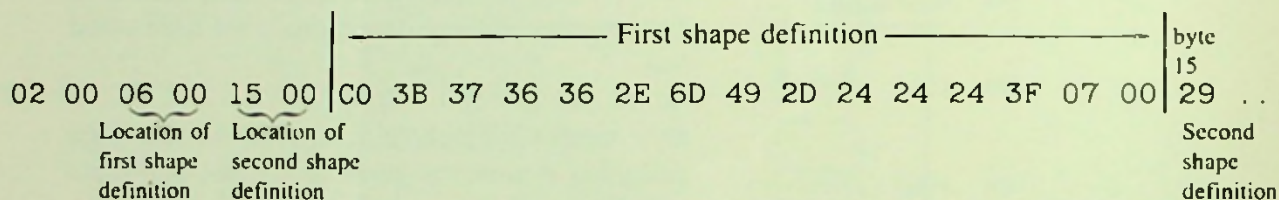
### Example

Suppose the shape of Example 5 (the [ ] shape at the beginning of the chapter) is the only shape in the shape table. Then the complete shape table would look like this.



### Example

Suppose we had another shape in our table. The complete shape table might look like



Notice that the location of the first shape definition is now 0006, not 0004. This is so because the shape definition comes after the index and the index is 2 bytes longer in this example than in the first example. In general, the more shape definitions are stored in a shape table, the larger the locations of the shape definitions will be. The location of the second shape definition is 0015 in this example because the index takes up \$06 bytes and the first shape definition is \$0F bytes long ( $\$06 + \$0F = \$15$ ).

## Entering the Shape Table into Apple's Memory

The locations of the shape definitions in the shape table are not the same as memory locations in Apple's memory. The location of a shape definition gives the definition's position relative to the first byte in the shape table. The table (which consists of the index and the shape definitions) itself should be stored beginning in memory location \$1E00. This location was chosen to avoid interference by Applesoft programs, high-resolution graphics, DOS, and the like. Shape tables do not *have to* be loaded beginning in \$1E00. Any



out-of-the-way block of memory will do. The shape table index and shape definitions can be entered most easily into the memory locations you choose by use of the monitor.

When you use the command DRAW to draw the shape, Applesoft has to know in which address the first byte of your shape table appears. The address of the first byte of the shape table (which usually will be \$1E00) is stored in locations \$E8 and \$E9. Place the least significant byte of the address in \$E8 and the most significant byte of the address in \$E9. Again it is easiest to enter the shape table's address into locations \$E8 and \$E9 by using the monitor.

It should be mentioned that you can BSAVE a shape table just as with any other data. For example,

```
BSAVE SHAPE, A$1E00, L$53
```

could be used to save a shape table called SHAPE. The table's first byte is stored in \$1E00 and the table is \$53 bytes long.

```
BLOAD SHAPE
```

could be used to load the shape table back into memory, beginning in location \$1E00. If you use BLOAD to load a shape table into memory, do not forget to enter the shape table's address into \$E8 and \$E9. The BLOAD command will not reload \$E8 and \$E9 for you.

### **Using Shape Tables—DRAW, XDRAW, ROT, and SCALE**

The BASIC Commands DRAW, XDRAW, ROT, and SCALE are used in Applesoft programs for manipulating high-resolution shapes.

#### **The ROT Command**

The syntax of the command is

```
ROT= expr.
```

where *expr.* is any arithmetic expression legal in Applesoft. The value of the expression must be between 0 and 255 (decimal) inclusive. The ROT command can be used in the immediate or the deferred execution mode. You use ROT to reproduce your shape rotated in any orientation you choose. ROT=0 is used with DRAW to reproduce the shape as originally defined. Each increment of 16 in the value stored in ROT causes the shape to be rotated 90 degrees clockwise. For example, set ROT = 16 if the shape is to be drawn rotated 90 degrees clockwise, set ROT = 32 if the shape is to be drawn rotated 180 degrees clockwise, and set ROT = 48 if the shape is to be drawn rotated 270 degrees clockwise. If ROT = 64 is used, the shape will be drawn as defined since a rotation of 360 degrees rotates the shape back to its original orientation. The pattern of clockwise rotations repeats itself for

values of ROT greater than 64. Values of ROT less than 16 will cause rotations of less than 90 degrees. For example, ROT = 8 is used if a shape is to be rotated 45 degrees clockwise. For rotations of less than 90 degrees to be recognized as valid, the SCALE (see the next section for a discussion of the SCALE command) of the shape must be greater than one. If a SCALE of one is used, then the only values of ROT recognized are multiples of 16. (This rule is true in general, but there are exceptions. The values of ROT recognized depend on the shape being rotated as well as the SCALE.)

In general, the larger the scale, the more values of ROT with different effects are available. The best way to discover the effect of a particular value of ROT in combination with a particular SCALE is to experiment with your shape.

### The SCALE Command

Normally plotting vectors will plot one point and then move. By using

SCALE= expr.

(where expr. is any valid arithmetic expression whose value is between 0 and 255), you can enlarge your shapes. SCALE can be used in the immediate or the deferred execution mode. If

SCALE=1, then each plotting vector plots just one point before moving.

SCALE=2, then each plotting vector plots two points (in the direction of the arrow) and then moves.

SCALE=3, then each plotting vector plots three points and then moves.

If SCALE is set equal to zero, the maximum enlargement is produced. Thus SCALE=0 is really equivalent to a scale of 256. A SCALE factor  $k$ , then, is simply used to enlarge the shape  $k$  times.

You should not be surprised if your shape becomes distorted if it is "blown up" by using a scale factor. Consider the left part of the bracket-shaped design for which we earlier constructed a shape table. If you look at the vectors used to plot the top part of the bracket, you will see that  $\rightarrow \rightarrow \uparrow$  were used. To plot the bottom part of the left bracket, we used  $\leftarrow \leftarrow \leftarrow$ . Suppose we use a scale factor of 3 to blow up this shape. The top part of the bracket originally was drawn by plotting three dots. The enlarged version will not have nine dots, however. This is so because the last arrow that is used to plot this part of the bracket (the  $\uparrow$  arrow) points downward. Therefore the three dots that the  $\uparrow$  vector will contribute to the enlarged shape will not extend the top part of the bracket leftward. The three dots will be oriented vertically. On the other hand, all three arrows used to plot the bottom part of the bracket point to the right. Therefore the bottom part of the bracket will be a full nine dots long. To summarize, when our shape is plotted with SCALE=1, it will appear exactly as we defined it:  $\left[ \begin{array}{c} \rightarrow \rightarrow \uparrow \\ \leftarrow \leftarrow \leftarrow \end{array} \right]$ . When SCALE=3 is used, it will be slightly distorted and look like  $\left[ \begin{array}{c} \rightarrow \rightarrow \uparrow \\ \leftarrow \leftarrow \leftarrow \end{array} \right]$  (distortion exaggerated for the sake of clarity). The larger the scale factor you use, the worse the distortion will be.



### The DRAW Command

The DRAW command is used in Applesoft after setting the high-resolution graphics mode (use the HGR command), the color for the shape (use HCOLOR = expr.), the ROTation value, and the SCALE factor. Of course, a shape table must be in memory also. The syntax of the DRAW command is

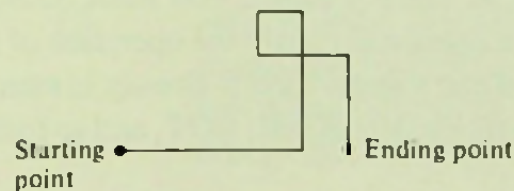
DRAW expr. 1 AT expr. 2, expr. 3

This expression tells Apple which shape definition in your shape table to draw. The value of the expression cannot exceed the number of shape definitions in your shape table.

The starting point (the coordinate of the dot that is plotted by the first plotting vector in the shape definition) will be the high-resolution point whose X coordinate (or column) is expr. 2 (the value of this expression must be between 0 and 279 inclusive) and whose Y coordinate (or row) is expr. 3 (the value of this expression must be between 0 and 191 inclusive).

The DRAW command can be used in the immediate or the deferred execution mode. You should take some care in selecting the starting point for your shape. Obviously you should avoid plotting your shapes too close to the edge of the screen so that there will be room to display your entire shape.

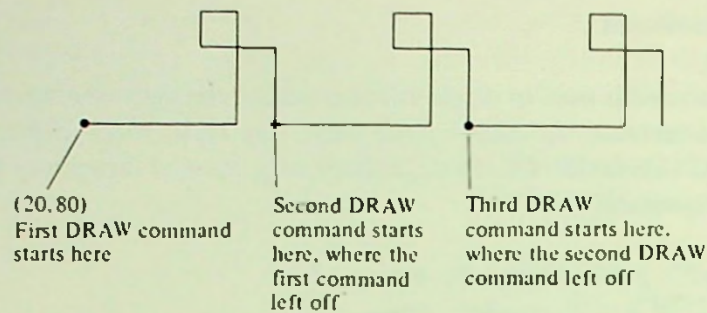
The DRAW command can be used to produce "chains" of a given shape. If DRAW expr. is used (without the AT expr. 2, expr. 3 part), then the shape definition will be DRAWn starting at the last point plotted by the previous DRAW (or XDRAW or HPLOT) command. For example, suppose we have a shape definition for the shape



stored as the first shape definition in a shape table. The Applesoft program

```
10 HGR
20 HCOLOR = 1
30 ROT = 0
40 SCALE = 1
50 DRAW 1 AT 20, 80
60 DRAW 1
70 DRAW 1
```

would produce this output:



### The XDRAW Command

The command

```
XDRAW expr. 1 AT expr. 2, expr. 3
```

works exactly as the DRAW command does, except that the shape is drawn in the complement of the color that already exists at each point being XDRAWn. This means that XDRAW can be used to erase a shape already DRAWn. To erase a shape, simply XDRAW over it (that is, use the same starting coordinates in the XDRAW command that you used when the shape was plotted using DRAW). Thus the whole purpose of the XDRAW command is to give you an easy way to erase a shape already DRAWn without clearing the entire screen of graphics. The XDRAW command is available with the same chaining option that DRAW had (to *chain* shapes, leave off the AT expr. 2, expr. 3 part of the statement).

For XDRAW to have its erasing effect, you must XDRAW over a shape already DRAWn. The following examples will clarify the operation of the XDRAW command. In these examples, it is assumed that a shape table is already in memory. Details such as setting the high-resolution graphics mode, HCOLOR, ROT, and so forth, are also assumed to have been taken care of.

#### Example 1:

The shape is DRAWn and then XDRAWn. The result is a blank screen.

```
80 DRAW 1 AT 10,10
90 XDRAW 1 AT 10, 10
```



**Example 2:**

```

      .
      .
      .
80  XDRAW 1 AT 10, 10
      .
      .
      .

```

Since the shape is not being XDRAWn over an already existing shape, it will be plotted as usual.

**Example 3:**

```

      .
      .
80  XDRAW 1 AT 10, 10
90  XDRAW 1 AT 10, 10
      .
      .
      .

```

The first XDRAW command draws the shape on a blank screen, as in Example 2. The second XDRAW command will XDRAW over this shape, and the result will be a blank screen.

**Example 4:**

```

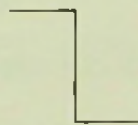
      .
      .
      .
80  XDRAW 1 AT 10,10
90  DRAW 1 AT 10, 10
      .
      .
      .

```

The XDRAW command plots the shape as in Example 2. If you DRAW over this shape, you merely replot what already exists on the screen. The DRAW command will not erase if you DRAW over an already existing shape. This means "an XDRAW command cancels a DRAW command" only if the XDRAW command is executed after the DRAW command.

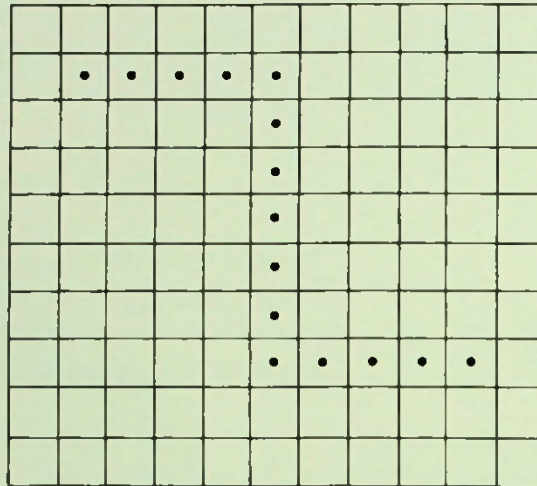
**Step-by-Step Example: Constructing and Using a Shape Table**

We close this chapter by giving a step-by-step description of the construction of a shape table for this shape:

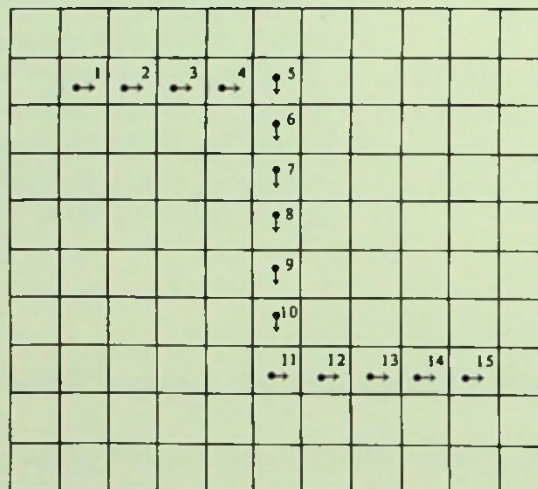


Applesoft programs using the completed shape table and the commands DRAW, XDRAW, ROT, and SCALE will be given.

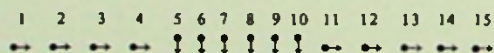
First let us construct the shape definition. We begin by plotting the shape on a grid:



Next define the shape in terms of plotting vectors.

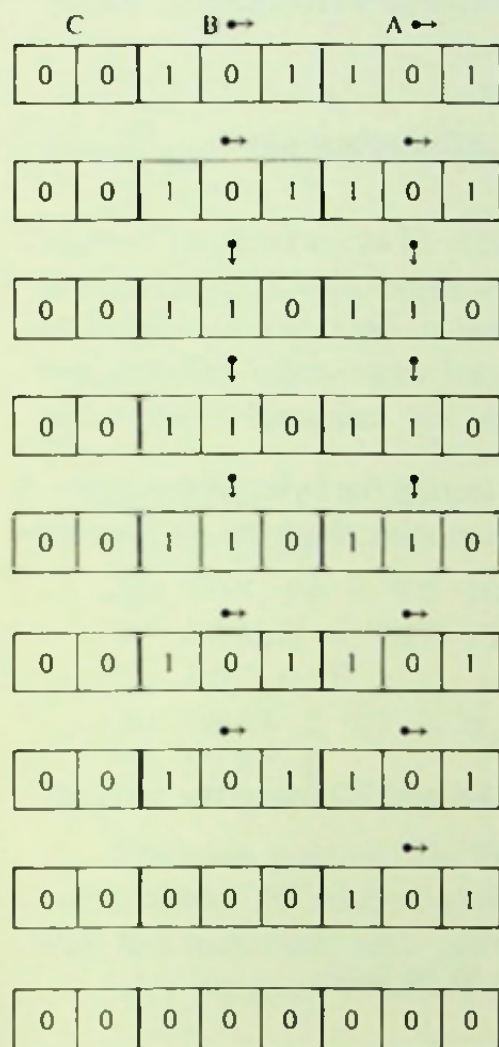


If we choose the upper left-hand corner as our starting point, the defining plotting vectors become





Next place the codes for the plotting vectors in the bytes of the shape definition:



The vectors are coded in the order in which they were used to trace out the shape (code vector 1 first, vector 15 last). The vectors are placed in the bytes working from right to left.

A byte filled with zeros marks the end of the shape definition.

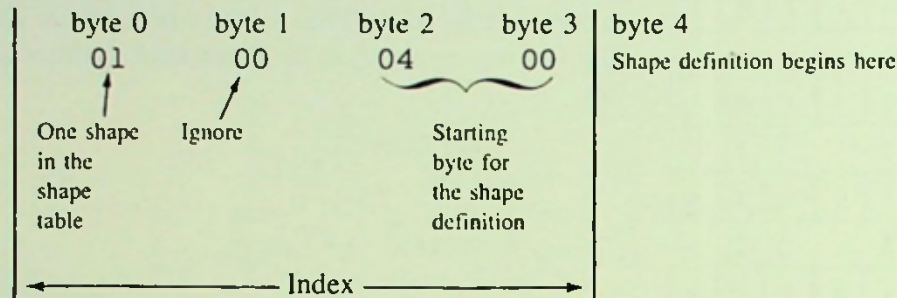
The bits are now converted to their HEX equivalents.

| <i>Bits Grouped by<br/>Sections A, B, C</i> | <i>Bits Grouped<br/>by Fours</i> | <i>HEX Equivalents</i> |
|---------------------------------------------|----------------------------------|------------------------|
| 00 101 101                                  | 0010 1101                        | 2D                     |
| 00 101 101                                  | 0010 1101                        | 2D                     |
| 00 110 110                                  | 0011 0110                        | 36                     |
| 00 110 110                                  | 0011 0110                        | 36                     |
| 00 110 110                                  | 0011 0110                        | 36                     |
| 00 101 101                                  | 0010 1101                        | 2D                     |
| 00 101 101                                  | 0010 1101                        | 2D                     |
| 00 000 101                                  | 0000 0101                        | 05                     |
| 00 000 000                                  | 0000 0000                        | 00                     |

Therefore the bytes of the shape definition are

2D 2D 36 36 36 2D 2D 05 00

We are now ready to add the index to the shape definition in order to complete the shape table. The correct index will be



To enter the shape table, we use the monitor. Begin storing the bytes of the index in location \$1E00. After the bytes of the index are stored, you store the bytes of the shape definition.

```
*1E00:01 00 04 00 2D 2D 36 36
*1E08:36 2D 2D 05 00
```

Next the starting address of the table is put in locations E8 and E9 using the monitor:

```
*E8:00 1E
```

The shape table is now in Apple's memory and is ready to be used.

### Motion Graphics—Program 1

You can create an illusion of movement by drawing a shape, erasing it, and then redrawing it in a slightly different location.

```
10 HGR
20 HCOLOR = 3
30 ROT= 0
40 SCALE = 3
50 FOR I= 10 TO 250
60 DRAW 1 AT I,80
70 FOR J= 1 TO 50
80 NEXT J
90 XDRAW 1 AT I,80
100 NEXT I
```

You can slow down or speed up the shape's movement by increasing or decreasing the upper limit on the J loop (lines 70 and 80).

It should be mentioned that this method of programming motion graphics is far from



perfect. If you try to make the shape move too fast, the picture will have a "flickering" quality. For moderate or slow rates of motion, the quality of the output is good, however.

### Computer Art—Program 2

Interesting patterns can be produced by rotating shapes and changing the scale factors used to plot shapes. Large shapes that "stick out past the edge of the screen" will "wrap around" and appear on the opposite side of the screen. This effect leads to some rather remarkable designs. The following two programs vary the SCALE and ROTation used to plot our shape and produce computer "art" as a result.

```
10 HGR
20 HCOLOR = 3
30 FOR I= 1 TO 100
40 SCALE = INT (I/5) + 1
50 ROT = I
60 DRAW 1 AT 135, 80
70 NEXT I
```

The scale is set equal to  $\text{INT}(I/5) + 1$  on line 40 instead of  $\text{INT}(I/5)$  to avoid a SCALE factor of zero. SCALE=0 gives the maximum magnification of the shape. Patterns drawn with this maximum scale clutter up the screen too quickly.

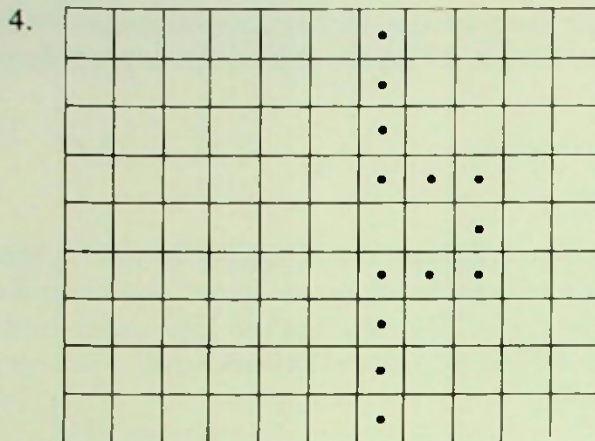
This program combines translation with rotation to produce a pattern:

```
10 HGR
20 HCOLOR = 3
30 FOR I= 1 TO 80
40 SCALE = INT (I/5) + 1
50 ROT = I
60 DRAW 1 AT 3*I, 80
70 NEXT I
```

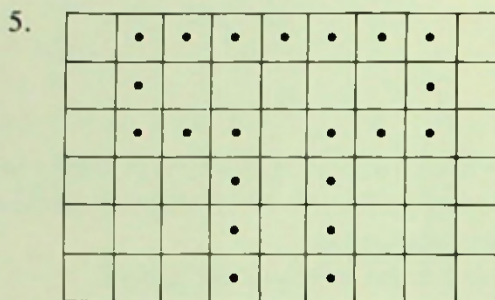
You now have the knowledge needed to do some fairly sophisticated work with both sound and graphics. It is in both these areas that machine language truly unlocks the power of the Apple.

### Review Questions

1. The two main parts of a shape table are \_\_\_\_\_ and an \_\_\_\_\_.
2. A shape table can hold shape definitions for up to \_\_\_\_\_ shapes (answer in decimal).
3. The last byte of a shape definition has three sections filled with \_\_\_\_\_.



Calculate the HEX numbers making up the shape definition of the shape shown. Let the top dot be the starting point of the shape.



Calculate the HEX numbers making up the shape definition of the shape shown. Let the dot in the lower left be the starting point for the shape.

6. A shape table has three shape definitions in it. The first shape definition is eight bytes long, the second is seventeen bytes long, and the third is twenty-two bytes long. Construct the index for this shape table.
7. Construct a complete shape table that contains the two shape definitions calculated in questions 4 and 5.
8. ROT = 16  
is used to rotate a shape \_\_\_\_\_ degrees clockwise.
9. Use SCALE = \_\_\_\_\_ to reproduce your shape in its original size.
10. Give the DRAW command that would be used to reproduce the second shape in a shape table starting at the point (80,90).
11. Explain what the command

DRAW 3

would do.



**TRUE or FALSE**

12. Plotting vectors can represent only up, down, right, or left moves. Diagonal moves cannot be represented.
13. No vector that plots a point before moving has a two-digit binary code.
14. Each byte of a shape definition can hold the codes for up to two plotting vectors.
15. The leftmost section of a byte in a shape definition (section C) can never hold the code for a vector that plots a point before moving.
16. The index of a shape table that contains five shape definitions is 10 bytes long.
17. The index of a shape table is placed before the shape definitions.
18. The first few bytes of a shape table are

0A 00 16 00 . . .

The first shape definition is stored beginning in memory location

\$0016

19. Shape tables cannot be stored on a disk using the same commands that were used to save binary programs.
20. The SCALE factor used to DRAW a shape affects the number of ROTation values recognized by Applesoft.
21. SCALE = 0  
reduces your shape to a single point.
22. The output of this program segment would be a blank screen.

```
80 XDRAW 1 AT 80,90
90 DRAW 1 AT 80,90
```

23. The output of this program segment would be a blank screen.

```
80 XDRAW 1 AT 80,90
90 XDRAW 1 AT 80,90
```

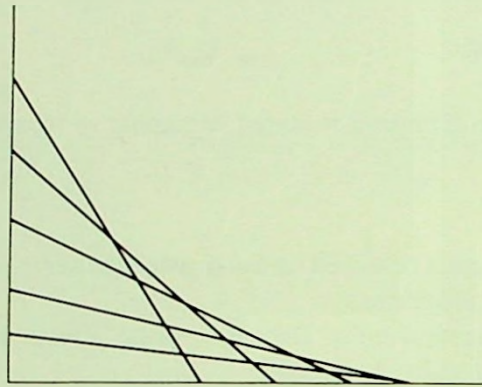
**Programming Problems**

The programs listed here are only suggestions. You are encouraged to experiment with shapes, motion graphics, and computer art of your own design.

1. Select several letters of the alphabet. Construct a shape table containing the shape definitions for these letters. Use your shape table to print the letters in various locations on Apple's screen.
2. Construct a shape table containing the shape definition for the shape ↑. Use this shape table to simulate the motion of the hour and minute hands of a clock.

Program the minute hand to complete a revolution 12 times faster than the hour hand does, just as on a real clock. Your clock's hands should rotate faster than the hands of a real clock, however, so that you can see your program working.

3. Try simulating the pattern produced when several stones are thrown into a pond. This can be done by drawing circles with fixed centers but increasing radii.
4. Use the chaining effect of DRAW to produce a fancy border around the edge of the screen such as those found on some stock certificates.
5. If a multiple-exposure photograph of a ladder sliding down the side of a building were taken, the result might look like



See if you can program the Apple to produce a design similar to the one on this multiple-exposure photograph by using a shape table.



## ***Chapter 8***

# ***Indirect Addressing and Some Miscellaneous Topics***

### **Indirect Addressing**

Suppose you have five programs in memory, beginning at locations \$900, \$A00, \$B00, \$C00, and \$D00. A short program beginning at location \$300 prompts the user to input a number from one to five. This “menu” program is supposed to branch to program 1, 2, 3, 4, or 5, depending on the input. One approach to writing the menu program would be to branch to a JMP statement that selects the proper program. The input from the keyboard would determine which JMP command is branched to. Consider the following program segment:

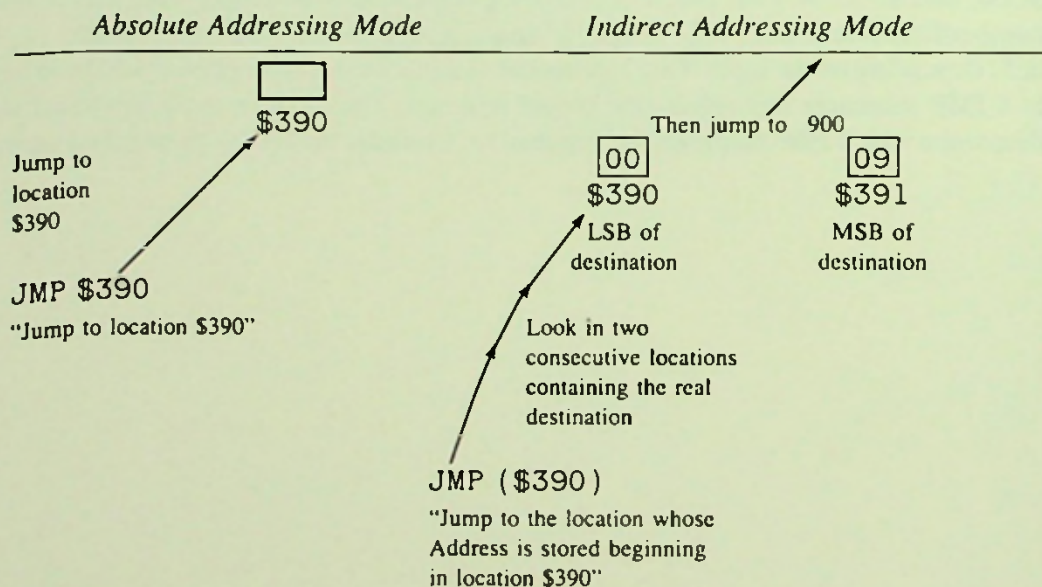
*Rough Assembly Language Version**Finished Assembly Listing*

|          |                |            |
|----------|----------------|------------|
| JSR FD35 | 0300- 20 35 FD | JSR \$FD35 |
| CMP #B1  | 0303- C9 B1    | CMP #\$B1  |
| BEQ ?    | 0305- F0 18    | BEQ \$031F |
| CMP #B2  | 0307- C9 B2    | CMP #\$B2  |
| BEQ ?    | 0309- F0 11    | BEQ \$031C |
| CMP #B3  | 030B- C9 B3    | CMP #\$B3  |
| BEQ ?    | 030D- F0 0A    | BEQ \$0319 |
| CMP #B4  | 030F- C9 B4    | CMP #\$B4  |
| BEQ ?    | 0311- F0 03    | BEQ \$0316 |
| JMP D00  | 0313- 4C 00 0D | JMP \$0D00 |
| JMP C00  | 0316- 4C 00 0C | JMP \$0C00 |
| JMP B00  | 0319- 4C 00 0B | JMP \$0B00 |
| JMP A00  | 031C- 4C 00 0A | JMP \$0A00 |
| JMP 900  | 031F- 4C 00 09 | JMP \$0900 |

The rest of the program is stored beginning in locations \$900, \$A00, \$B00, \$C00, and \$D00.

This kind of program structure becomes awkward, especially if the number of possible branches is large.

Another method of producing the desired branching involves the use of indirect addressing. In the absolute addressing mode, the operand of JMP gives the destination of the jump. In the indirect addressing mode, the operand of JMP gives a location in which the destination is stored. Schematically we have

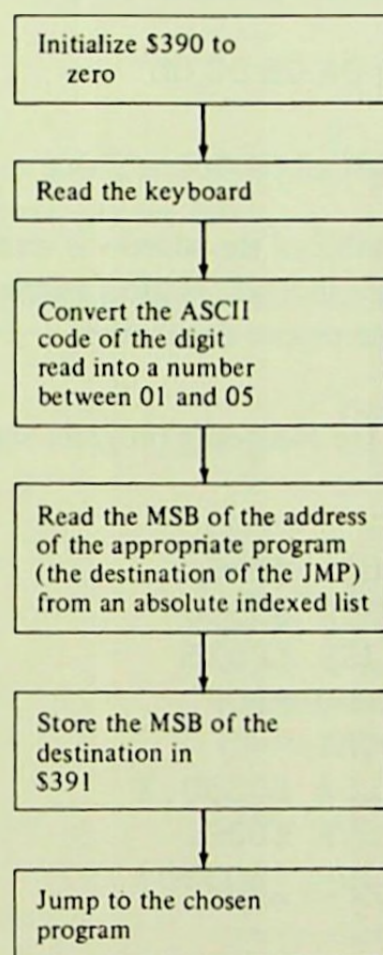




An indirect jump is thus a two-step process. The operand of JMP tells Apple where the real destination is stored. Once this destination is looked up in memory, then the jump to the real destination is made. Since addresses are four-digit HEX numbers, the location \$390 cannot contain the entire destination address. Rather it contains the least significant byte (LSB) of the final destination. The most significant byte (MSB) of the destination is assumed to be stored in the next higher location (\$391 in this example). Note that parentheses around the address \$390 are used to distinguish the indirect addressing mode (op code 6C) from the absolute addressing mode (op code 4C) in assembly language. The indirect addressing mode command JMP is 3 bytes long as the operand of JMP is a 2-byte address.

The following flowchart illustrates how indirect addressing could be used to solve the branching problem in the menu program.

#### Flowchart



Assembly Language    Comments

```
LDA #00
STA $390
```

Notice that since the LSB of the destination address is always 00, this byte could have been stored in \$390 before running the program by using the monitor.

```
JSR FD35
```

```
AND #0F
```

It is assumed that the MSB of the destination address is stored in a list in locations \$351 to \$355. This list should be loaded into memory before running the program by using the monitor command

```
TAX
LDA $350,X
```

\*351:09 0A 0B 0C 0D

```
STA $391
```

Once the MSB of the address is read from the absolute indexed list and is stored in \$391, indirect addressing can be used to execute the jump to the proper destination.

```
JMP ($390)
```

The finished assembly listing of the foregoing program segment is presented here for the sake of completeness:

|       |          |              |
|-------|----------|--------------|
| 0300- | A9 00    | LDA #\$00    |
| 0302- | 8D 90 03 | STA \$0390   |
| 0305- | 20 35 FD | JSR \$FD35   |
| 0308- | 29 0F    | AND #\$0F    |
| 030A- | AA       | TAX          |
| 030B- | BD 50 03 | LDA \$0350,X |
| 030E- | 8D 91 03 | STA \$0391   |
| 0311- | 6C 90 03 | JMP (\$0390) |

The rest of the program is stored beginning in locations \$900, \$A00, \$B00, \$C00, and \$D00.

### Preindexed Indirect Addressing Mode

Only the JMP command is available in the indirect addressing mode. Many other important machine language commands are available in both the preindexed indirect (also called the



indexed indirect addressing mode) and the postindexed indirect addressing mode (also called the indirect indexed addressing mode).

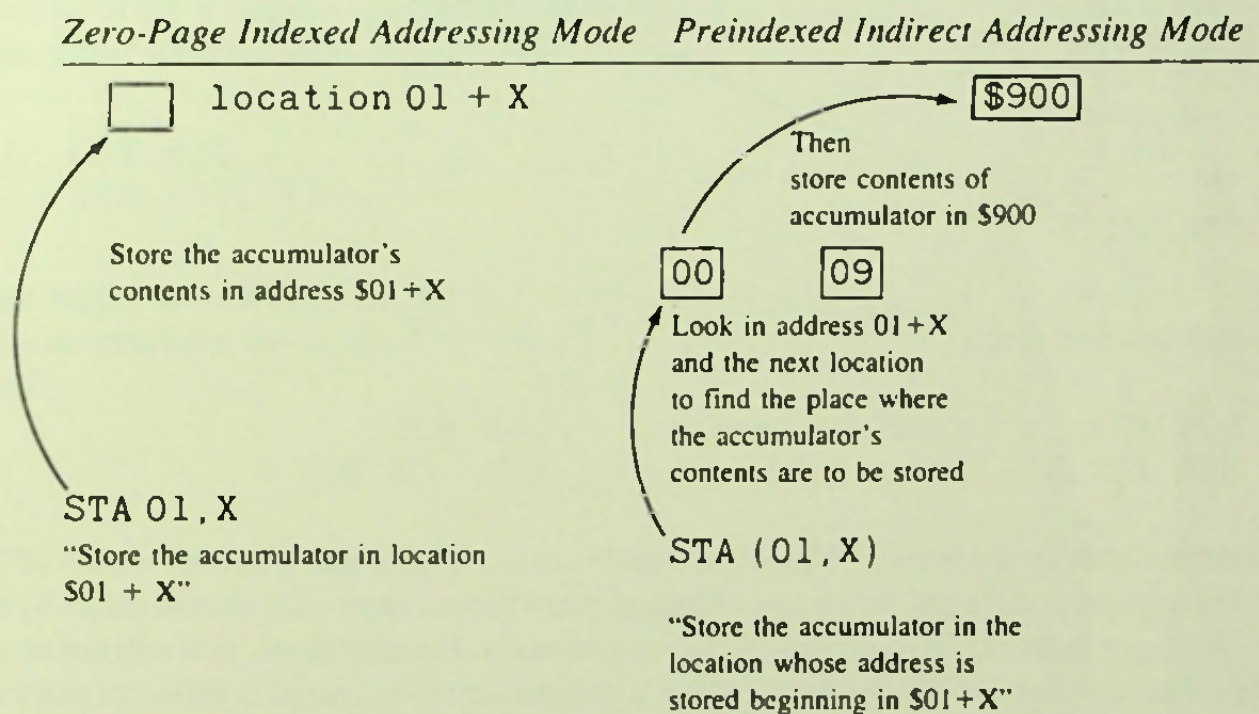
The preindexed indirect addressing mode is conceptually very close to the indirect addressing mode just discussed. Consider the instruction

`STA(01,X)`

This command works as follows: First the value stored in the X register is added to the "base address" \$01. The resulting sum tells the computer which *zero page* address to examine to find the least significant byte of the address where the accumulator's contents will be stored. The most significant byte of the destination address is assumed to be in the next higher zero-page location. For example, the two commands

`LDX#01`  
`STA (01,X)`

tell the Apple to look in locations \$0002 and \$0003 for the *address* where the accumulator's contents will be stored. Schematically we have



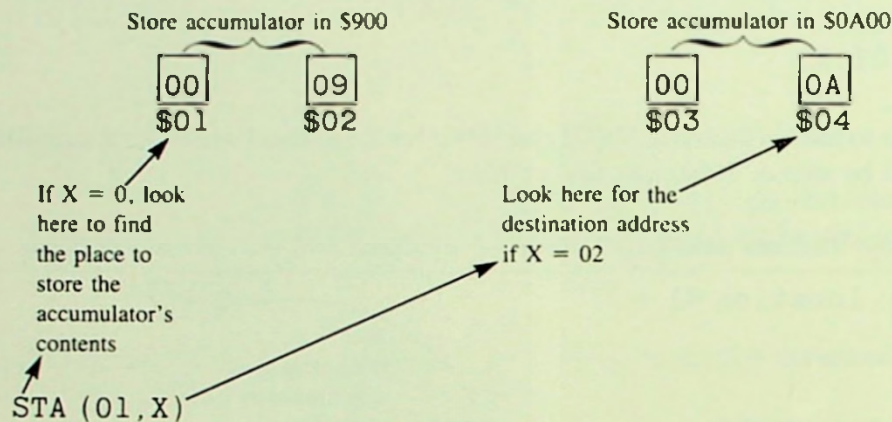
It is worth emphasizing that the operand in the preindexed indirect addressing mode always points to a zero-page location. This was not the case in indirect addressing mode. Preindexed commands are thus 2 bytes long, since the "base address" is a (1-byte) zero-page address. The machine language form of the assembly command

|                        |    |      |                   |
|------------------------|----|------|-------------------|
| <code>STA(01,X)</code> | is | 81   | 01                |
|                        |    | ↑    | ↑                 |
|                        |    | op   | operand is a      |
|                        |    | code | zero-page address |

We have used \$01 as a base address. Any other unused zero-page address is acceptable as an operand, although most are already used by Apple for other purposes. You should be careful about using other zero-page addresses in your indirect indexed addressing mode commands (the first few zero-page locations are safe). For example, do not use location \$4E or \$4F as a base address because those locations are filled with random numbers by the FD35 subroutine.

You should also note that although the form STA(01,Y) would seem to be perfectly logical, it is *not* a valid assembly language command. Preindexed indirect addressing always makes use of the X register as its index.

By changing the X register's value in STA(01,X), you can tell Apple to look in different locations for the final storage location of the accumulator's contents. Schematically we have



It should be clear that

|            |                  |            |
|------------|------------------|------------|
| LDX #02    | is equivalent to | LDX #00    |
| STA (01,X) |                  | STA (03,X) |

(Both commands do the work of STA \$0A00 in this case.) Notice that if several values of X are used to point to different zero-page addresses, then those values of X should differ by at least two to prevent "overlap" of the destination addresses. In many cases, you will not need to use different values of X if the accumulator's contents are to be stored in different places. To change storage destinations, you can keep X = 0 but reload locations \$01 and \$02 with the new destination address before executing

```
LDX #00
STA (01,X)
```



Finally, note how parentheses are used around the operand (01,X) to distinguish between the preindexed indirect addressing mode (op code 81) and the zero-page indexed addressing mode (op code 95) in assembly language.

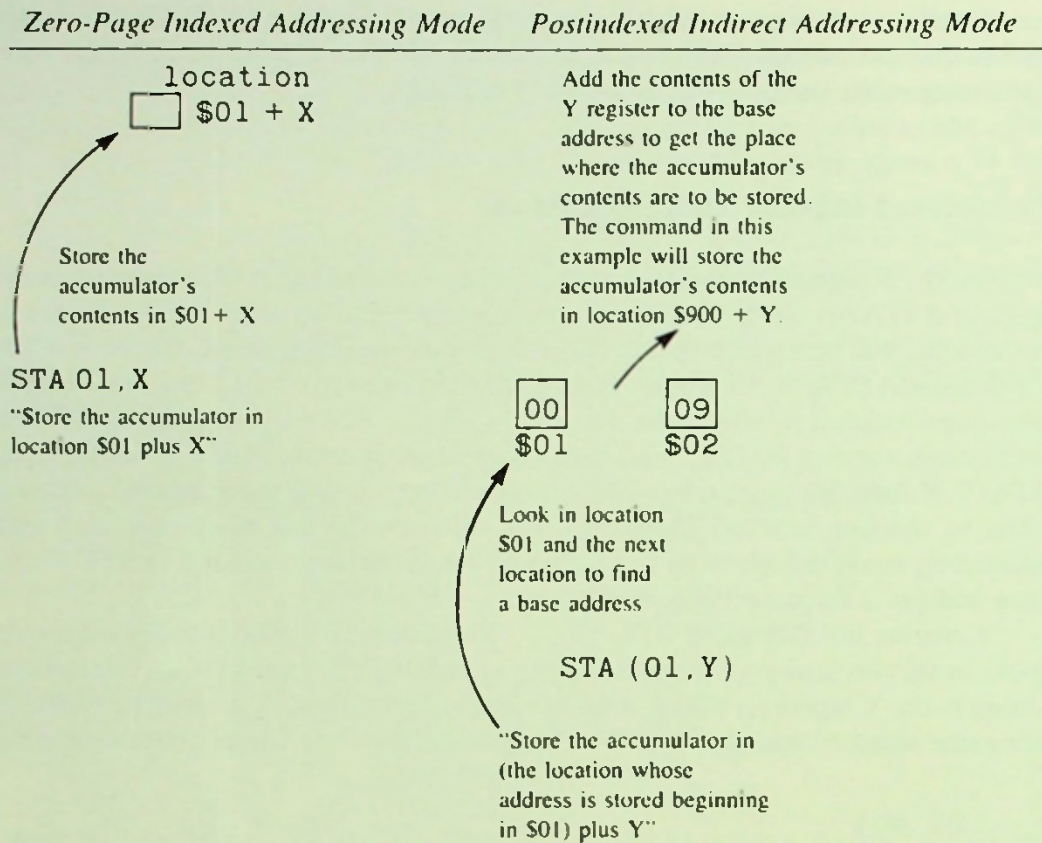
### **Postindexed Indirect Addressing Mode**

Varying the X register has a different effect on the command STA 01,X than it does on the command STA (01,X). If STA 01,X is put in a loop using the X register as a counter, then information will be stored in consecutive memory locations beginning with location \$01. If the command STA(01,X) is used, then varying the contents of the X register changes the zero-page location in which the destination address is found. In postindexed indirect addressing, varying the index will store information in consecutive memory locations as STA 01,X does. We can thus think of postindexed indirect addressing as being conceptually close to absolute indexed addressing. The difference is that the postindexed indirect addressing mode will allow us to read the base address from memory. In STA 01,X, the base address is built into the command as an operand.

Consider the instruction STA (01),Y. The command works as follows: First Apple looks in the two zero-page locations beginning at \$01 for a "base address." Next the value stored in the Y register is added to the base address just found. The resulting sum tells the computer where to store the contents of the accumulator. For example, the two commands

```
LDY #01  
STA (01),Y
```

tell Apple to look in locations \$0001 and \$0002 for a base address B. Then the contents of the accumulator are stored in the address B + Y. This situation can be represented schematically:



The operand in postindexed indirect addressing always "points to" a zero-page location. Therefore postindexed addressing mode commands are 2 bytes long, just as preindexed addressing mode commands were. The machine language form of the assembly language command

|             |    |                 |                                        |
|-------------|----|-----------------|----------------------------------------|
| STA (01), Y | is | 91              | 01                                     |
|             |    | ↑<br>op<br>code | ↑<br>operand is<br>a zero-page address |

Before going on, make sure you understand the difference between the use of the index in preindexed indirect addressing and postindexed indirect addressing. In preindexed indirect addressing, the X register is added to the command's operand to locate the zero-page addresses that contain the command's destination address. In the postindexed indirect addressing mode, the zero-page addresses containing a base address are given by the operand of the command. The value of the Y register is added to this base address to obtain the command's destination address. It is worth emphasizing that the postindexed indirect addressing mode must use the Y register as its index. The command STA (01), X is invalid.



As an example, consider the program segment:

```
350- LDY #00
352- INY
353- STA (01).Y
355- CPY #05
357- BNE 352
```

If the contents of location \$01 are 00 and the contents of \$02 are 09, then this program segment will store the contents of the accumulator into locations 901, 902, 903, 904, and 905. The command STA (01).Y is thus equivalent to STA 900,Y in this particular case. You should see that use of the monitor command

```
*01:00 12
```

would make the command STA (01).Y equivalent to STA 1200.Y. Thus in the postindexed indirect addressing mode, the base address can be changed by putting different data into locations \$01 and \$02. In the absolute indexed addressing mode, the command itself must be changed to change the base address.

Finally, note the way parentheses are used around the operand 01 to distinguish the postindexed indirect addressing mode from the zero-page indexed addressing mode in assembly language.

We now have covered all the addressing modes available in Apple machine language. We close the section on indirect addressing modes by giving sample programs illustrating the use of postindexed indirect addressing and preindexed indirect addressing.

## Representation of Two-Dimensional Matrices

A matrix is an array of numbers. Each row of the matrix contains the same number of elements. The elements in each row are lined up to produce the columns of the matrix. A typical matrix might look like this.

|   |    |    |    |
|---|----|----|----|
| 5 | 8  | 9  | 3  |
| 7 | 14 | 17 | -3 |
| 0 | 9  | 2  | 8  |

We refer to any element in the matrix by giving its row and column numbers (called row and column indexes or subscripts). An example of a matrix with which you are familiar is the set of memory locations corresponding to the squares on the text screen. To review, recall that we can refer to any location on the text screen by giving a row number (rows will be labeled 0 to 23 for convenience in this section) and a column number (0 to 39). In machine language, each row R of the text screen has a starting address (see the text screen map in

Appendix F), which corresponds to the text screen square in row R, column 0. If you wish to find the address corresponding to a column other than column 0, you use the rule

$$\text{Address of row R, column C} = \text{Address of row R, column 0} + C$$

Thus each square of the text screen has an address that we can find if we know the row and column numbers of the square. The set of addresses of all the squares of the text screen can be thought of as forming a matrix with 24 rows and 40 columns (the first column is labeled 0 and the 40th is labeled \$27).

A knowledge of the addresses corresponding to any square on the screen is useful. If an ASCII code is stored in the memory location corresponding to row R and column C, then the character corresponding to the ASCII code will appear in row R, column C. In this section, we will write a program that allows us to display a character in row R, column C. The row number R and column number C will both be entered from the keyboard.

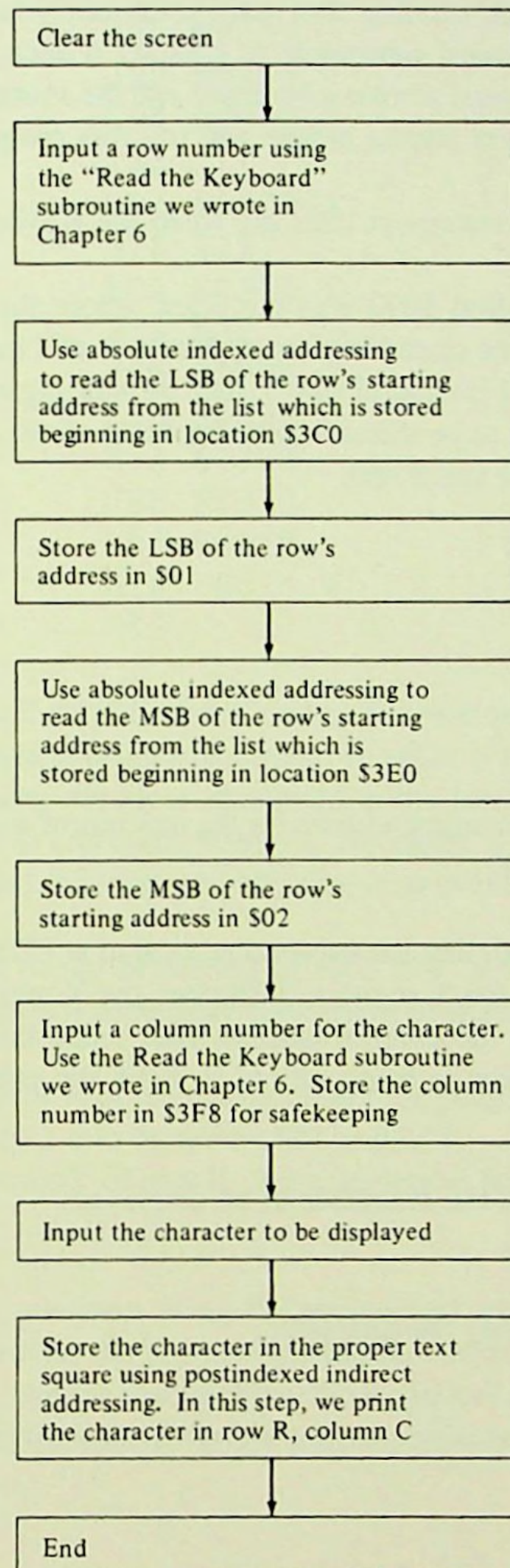
If we can find the address corresponding to row R, column C, then the program will be easy to write. Unfortunately there is no regular pattern to the addresses corresponding to the text screen squares. The addresses of adjacent squares in the same row are consecutive HEX numbers, but the addresses do not follow a regular pattern as you move down a column of the text screen. As a first attempt, it would seem logical to use a command such as STA 400,Y to store the ASCII code in the Yth column (of the first row). The difficulty here is that the starting address for each row is different, so that 24 different STA commands would be required if absolute indexed addressing were used to write this program. The solution, of course, is to use postindexed indirect addressing, since this will allow us to change the base address in the STA command.

We begin by storing the first address of each row in memory. Once the starting address of each row is available in storage, we will be able to take advantage of the power of postindexed indirect addressing. Beginning in location 3C0, we will place the least significant byte of the starting address for each row. Beginning in location 3E0, we will place the most significant byte of the starting address for each row. You can enter the required data by using these monitor commands:

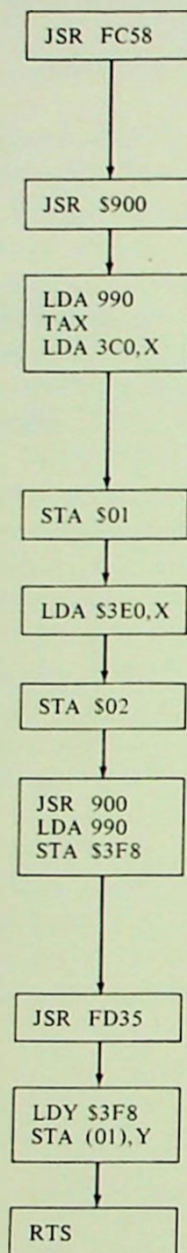
```
*3C0:00 80 00 80 00 80 00 80
*3C8:28 A8 28 A8 28 A8 28 A8
*3D0:50 D0 50 D0 50 D0 50 D0
*3E0:04 04 05 05 06 06 07 07
*3E8:04 04 05 05 06 06 07 07
*3F0:04 04 05 05 06 06 07 07
```



The following flowchart will be used to help us write the program:





*Assembly Language Comments*

Before running this main program, you first reload the Read the Keyboard subroutine of Chapter 6 back into memory. Notice that you must store a subroutine and the starting address for each row of the text screen before you run this program.

This statement calls the Read the Keyboard subroutine.

Location \$990 was the place where the Read the Keyboard subroutine stored its output. Notice that if the first row had been called row 1 instead of row 0, then the starting address for each row would have to be stored beginning in locations \$3C1 and \$3E1 instead of \$3C0 and \$3E0.

The starting address for the row is now stored in locations \$0001 and \$0002.

Recall that the subroutine located at FD35 (to be used next) scrambles the Y register. Therefore the Y register will have to be loaded with the column number after using this subroutine. The column number is being placed in \$3F8 for safekeeping.

Input the character to be displayed.

Display the character.



The completed assembly listing for the main program is shown here.

```
0300- 20 58 FC      JSR $FC58
0303- 20 00 09      JSR $0900
0306- AD 90 09      LDA $0990
0309- AA           TAX
030A- BD C0 03      LDA $03C0,X
030D- 85 01         STA $01
030F- BD E0 03      LDA $03E0,X
0312- 85 02         STA $02
0314- 20 00 09      JSR $0900
0317- AD 90 09      LDA $0990
031A- 8D F8 03      STA $03F8
031D- 20 35 FD      JSR $FD35
0320- AC F8 03      LDY $03F8
0323- 91 01         STA ($01),Y
0325- 60           RTS
```

You can use the method just illustrated on any list of numbers you wish to reference with two subscripts (row and column subscripts). In other words, you can represent any two-dimensional matrix using the ideas illustrated in the last program.

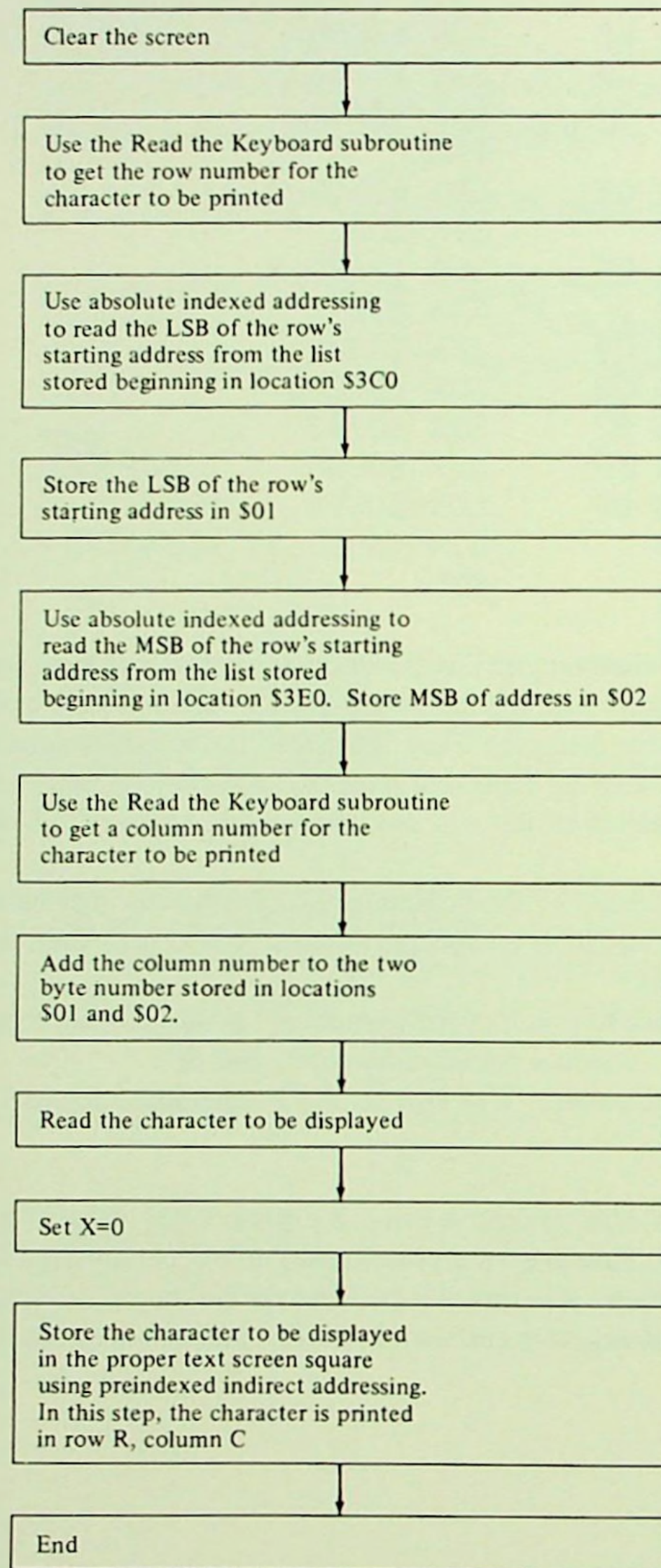
There are other ways to represent matrices. No claim is made that the method just presented is the most efficient. It is conceptually simple, however. The key steps used were:

1. Store the address of the beginning of each row in the matrix into consecutive memory locations. It is especially convenient to store each byte of the address in separate lists.
2. Any reference to row R of the matrix causes the program to store the Rth address on the lists of step 1 into locations \$01 and \$02.
3. To reference column Y of row R, use a statement such as STA (01),Y or LDA (01),Y.

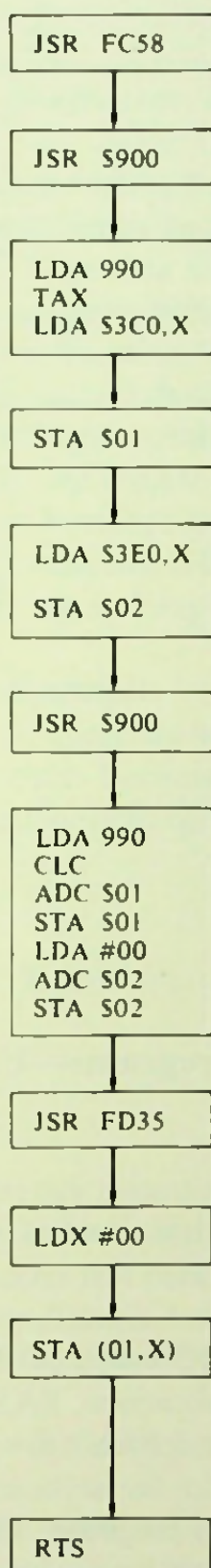
This program is now rewritten using the preindexed indirect addressing mode for comparison purposes. However, the postindexed indirect addressing mode is better suited for the foregoing problem. Assume that the Read the Keyboard subroutine of Chapter 6 and the lists of starting addresses for each row of the text screen have been loaded into memory.



## Flowchart





*Assembly Language Comments*

When you use preindexed indirect addressing, the actual destination address rather than a base address must be stored in zero-page memory. By adding the column number (loaded from location \$990) to the base address read from the lists stored beginning in \$3C0 and \$3E0, the actual destination address is computed.

The index X serves only to point to the location in zero-page memory where the destination address for the STA command has already been stored. In this step, the character is printed in row R, column C.

The completed assembly listing for the main program is shown here. You should study the preindexed and postindexed versions of the main program until you feel comfortable with the differences between these two indirect addressing modes.

```

0300- 20 58 FC      JSR $FC58
0303- 20 00 09      JSR $0900
0306- AD 90 09      LDA $0990
0309- AA           TAX
030A- BD C0 03      LDA $03C0,X
030D- 85 01         STA $01
030F- BD E0 03      LDA $03E0,X
0312- 85 02         STA $02
0314- 20 00 09      JSR $0900
0317- AD 90 09      LDA $0990
031A- 18           CLC
031B- 65 01         ADC $01
031D- 85 01         STA $01
031F- A9 00         LDA #$00
0321- 65 02         ADC $02
0323- 85 02         STA $02
0325- 20 35 FD      JSR $FD35
0328- A2 00         LDX #$00
032A- 81 01         STA ($01,X)
032C- 60           RTS

```

### **Integration of Machine Language Programs with BASIC Programs—Use of POKE, PEEK, and CALL**

The whole purpose for using machine language is to execute program segments that require faster than usual execution times. For example, the production of musical tones is not possible using BASIC alone, because a FOR-NEXT loop is not executed fast enough to produce the required number of speaker vibrations per second. Sort algorithms and advanced work with graphics also require faster speeds than BASIC provides. Many applications do not require exceptional speed, however. For these applications, BASIC is recommended. In fact, there are some situations in which it would not make sense to write a machine language program. For example, it is difficult to input data from the keyboard, do complex arithmetic calculations, or manipulate strings using machine language. On the other hand, BASIC is ideal for these applications. In this section, we learn how to integrate machine language subroutines into a BASIC program. Thus we will be able to farm out those parts of the program requiring great speed to machine language, while keeping the convenience and flexibility of BASIC for the rest of the program.



## The POKE Statement

Store

The

POKE expr. 1, expr. 2

statement stores the value of the expression expr. 2 (which must be between 0 and 255 inclusive) into the memory location whose address is given by expr. 1 (where expr. 1 and expr. 2 are any valid Applesoft expressions). POKE can be used in the immediate or deferred execution mode. Thus, POKE 768, 173 would store the value 173 (decimal) into memory location 768 (decimal). Note that the arguments of the statements POKE, PEEK, and CALL are decimal since all of Applesoft is based in the decimal number system. To use a POKE statement to enter data into memory, you first decide upon the HEX address in which you want to store the data. This HEX address must be converted to decimal. If the data to be stored are also given in HEX, you will have to convert these to decimal as well. Converting from HEX to decimal is done by "expanding" the HEX numeral as shown in the following and explained in Chapter 1.

### Example

Suppose we want to place the byte \$AD into memory location \$0300 using a POKE statement. Because the place value of each HEX digit is 16 times larger than the place immediately to the right, we have

$$\begin{array}{ccccccc}
 & 0 & & 3 & & 0 & & 0 \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 \text{Decimal equivalent of } \$0300 & = & 0 \times 16^3 & + & 3 \times 16^2 & + & 0 \times 16^1 & + & 0 \times 16^0 \\
 & = & 0 \times 4096 & + & 3 \times 256 & + & 0 \times 16 & + & 0 \times 1 \\
 & = & 768 & & & & & & 
 \end{array}$$

and

$$\begin{array}{ccccccc}
 & A & & D & & & \\
 & \downarrow & & \downarrow & & & \\
 \text{Decimal equivalent of } \$AD & = & 10 \times 16^1 & + & 13 \times 16^0 \\
 & = & 10 \times 16 & + & 13 \times 1 & = & 173
 \end{array}$$

Therefore the monitor command

\*300:AD     put AD into mem. loc 300

is equivalent to the BASIC statement

POKE 768, 173

location

value

It should be emphasized that even though the decimal form of the number to be stored must be used in a POKE statement, the HEX equivalent of the decimal number is what is actually stored in memory.

You can use POKes to enter data needed by a machine language subroutine from a BASIC program, but you should not use POKes to enter machine language programs themselves. In theory, you could enter the op codes and operands of your machine language program using POKes from BASIC. This job is best left for the monitor, however, since by using the monitor you avoid the HEX-to-decimal conversions just described.

### The PEEK Statement

As the name implies, the statement

```
PEEK (expr.)
```

is used to examine the contents of memory from within an Applesoft program. The expression *expr.* in parentheses can be any valid Applesoft expression. The parentheses are required. The address named by the expression must be the decimal equivalent of the address you wish to examine. The contents of memory will be displayed in decimal. PEEK can be used in the immediate or deferred execution mode.

#### Examples

Since \$300 = 768 in decimal

```
X=PEEK (768)
```

would store the decimal equivalent of the contents of memory location \$300 into the variable X.

The statement

```
PRINT 3*PEEK(768) + PEEK (769)
```

would calculate three times the contents of location \$300 (in decimal), add the contents of location \$301 (again in decimal), and print the decimal result.

### The CALL Statement

The statement

```
CALL expr.
```

causes the machine language subroutine stored beginning in the memory location named by the expression *expr.* to be executed. Again, *expr.* can be any valid Applesoft expression and the starting location of the machine language subroutine specified by *expr.* must be given in decimal. Thus, to execute a machine language subroutine stored in location \$300, we would use the statement

```
CALL 768
```



in our Applesoft program. CALL can be used in the immediate or deferred execution mode.

CALL works just as GOSUB does, except that the CALL statement executes a machine language subroutine and GOSUB executes an Applesoft subroutine. The RTS in a machine language subroutine works as the RETURN statement does in an Applesoft subroutine. The RTS statement will transfer control to whatever program called the subroutine. Transfer will be back to the statement immediately following the calling statement (CALL in Applesoft, JSR in a machine language program) or back to the monitor program itself if a Go command was used to run the subroutine directly from the monitor.

### Using POKE and PEEK to Work with Data Values Too Large to Fit into One Memory Location

Suppose we wish to CALL this subroutine from an Applesoft program:

#### *Assembly Listing*

|       |          |            |
|-------|----------|------------|
| 0300- | 20 58 FC | JSR \$FC58 |
| 0303- | A9 00    | LDA #\$00  |
| 0305- | 8D 94 03 | STA \$0394 |
| 0308- | 8D 95 03 | STA \$0395 |
| 030B- | 8D 96 03 | STA \$0396 |
| 030E- | 18       | CLC        |
| 030F- | AD 91 03 | LDA \$0391 |
| 0312- | 6D 93 03 | ADC \$0393 |
| 0315- | 8D 96 03 | STA \$0396 |
| 0318- | AD 90 03 | LDA \$0390 |
| 031B- | 6D 92 03 | ADC \$0392 |
| 031E- | 8D 95 03 | STA \$0395 |
| 0321- | AD 94 03 | LDA \$0394 |
| 0324- | 69 00    | ADC #\$00  |
| 0326- | 8D 94 03 | STA \$0394 |
| 0329- | 60       | RTS        |

#### Memory Use

|                              | <i>Most Significant Byte</i> | <i>Least Significant Byte</i> |       |
|------------------------------|------------------------------|-------------------------------|-------|
| First number<br>to be added  | \$390                        | \$391                         |       |
| Second number<br>to be added | \$392                        | \$393                         |       |
| Sum of the two numbers       | \$394                        | \$395                         | \$396 |

The reader may recognize this program as similar to the addition program that we wrote in Chapter 6. This program adds two four-digit HEX numbers together. The only difference between the program presented here and the version presented in Chapter 6 is that this program does not print the final answer as the original version did. It is our intention in this chapter to use this program as a subroutine. The machine language program listed here will perform the addition of two HEX numbers, but the answer will be printed by the Applesoft program that calls this subroutine. Therefore the Print the Answer commands stored in locations \$329 to \$33A of the original program were left out of this version.

In this section, we wish to write an Applesoft program that prompts the user for two numbers (decimal), stores the numbers into memory using POKE statements, executes the machine language subroutine listed to calculate the sum of the two numbers, and displays the result in decimal using PEEK commands.

The trickiest part of the program involves storing the numbers to be added into the memory locations \$390 to \$393. Consider the problem of storing the HEX equivalent of 3000 into locations \$390 and \$391. You might think that the statements

```
POKE 912, 30
POKE 913, 00
```

would get the job done (912 = \$390 and 913 = \$391). This is not the case, however, since

```
POKE 912, 30
```

stores \$1E in location \$390 and

```
POKE 913, 00
```

stores \$00 in location \$391. The decimal equivalent of \$1E00 is

$$\begin{array}{ccccccc}
 1 & & E & & 0 & & 0 \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 1 \times 16^3 & + & 14 \times 16^2 & + & 0 \times 16^1 & + & 0 \times 16^0 \\
 = 1 \times 4096 & + & 14 \times 256 & + & 0 \times 16 & + & 0 \times 1 \\
 = 7680
 \end{array}$$

To learn the correct way to store a decimal number larger than 255 into two consecutive memory locations using POKE commands, observe that each HEX digit in the most significant byte (MSB) is two places to the left of the digits in the least significant byte (LSB). Thus the place value of the MSB is  $16^2$ , or 256 times the place value of the LSB. This means that the value to be POKEd into the MSB can be found by dividing  $256_{10}$  into the number to be stored. The remainder of this division gives the value to be stored in the LSB. In this example,

$$3000 \div 256 = 11 \quad \text{with remainder } 184 \text{ (all numerals are in decimal)}$$



Thus the statements

```
POKE 912,11
POKE 913, 184
```

would correctly store the HEX equivalent of 3000 into locations \$390 and \$391. The reader should verify that the POKE commands store \$0B in location \$390 and \$B8 in location \$391, and that the HEX number \$0BB8 is equivalent to

$$11 \times 256 + 11 \times 16 + 8$$

or 3000. You should see that the Applesoft statements

```
10 INPUT "FIRST NUMBER ";N1
20 Q = INT (N1/256)      Q is the quotient obtained when N1 is divided
30 R = N1 - 256*Q        by 256 and R is the corresponding remainder.
40 POKE 912,Q
50 POKE 913,R
```

take care of the details of storing the HEX equivalent of the decimal number N1 into locations \$390 and \$391. We can also use the idea of place value to convert the final answer, which will be stored in locations \$394 through \$396, from HEX into decimal. We simply expand the HEX representation of the answer, remembering that the place value of each byte is 256 times the place value of the byte to the immediate right. Thus

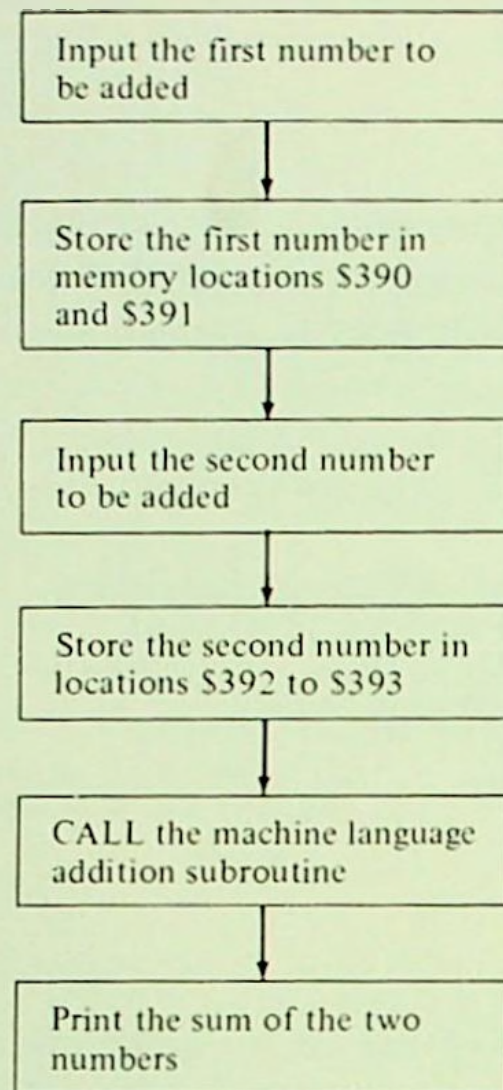
$$S = \underbrace{256 \times 256 \times \text{PEEK}(916)}_{\text{decimal equivalent of the contents of \$394}} + \underbrace{256 \times \text{PEEK}(917)}_{\text{decimal equivalent of the contents of \$395}} + \underbrace{\text{PEEK}(918)}_{\text{decimal equivalent of the contents of \$396}}$$

would store the answer to the addition problem in the variable S.

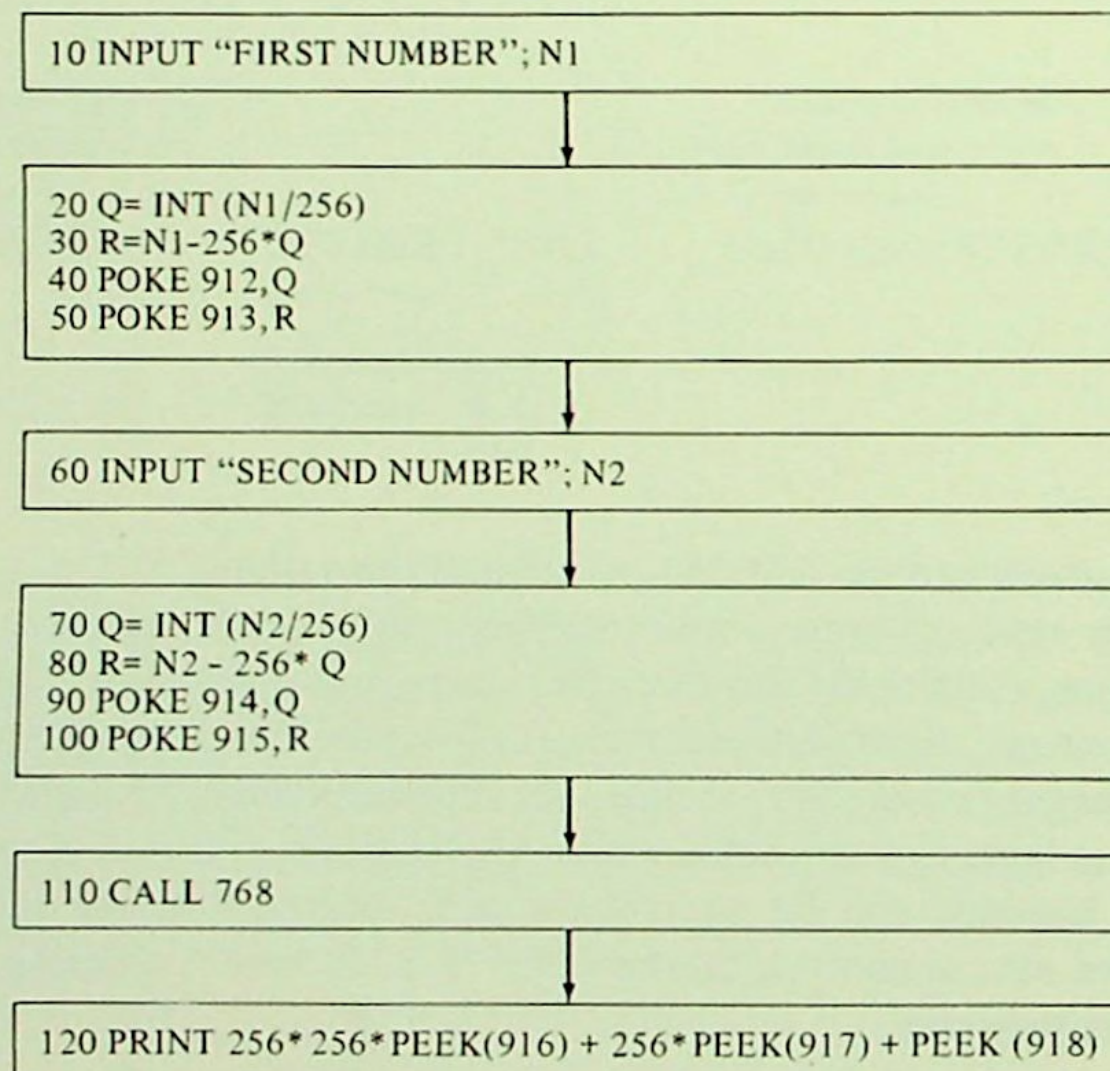
We are now ready to present an Applesoft program that could be used to call the addition subroutine. Admittedly this example is somewhat contrived. You normally would not do addition in an Applesoft program using a machine language subroutine. This example is presented because it is simple, yet illustrates many of the techniques used to integrate machine language subroutines into Applesoft programs in more complex situations. It will be assumed that the monitor has been used to store the addition subroutine previously shown into memory locations \$300 to \$329 before entering and running the Applesoft calling program.



## Flowchart



## Applesoft Program





Finally it should be mentioned that in some instances it may be necessary to set a value for HIMEM: to protect your machine language program from being overrun by your Applesoft program. The BASIC statement

```
HIMEM: expr.
```

sets the highest location available to BASIC equal to the value given by expr. (where expr. is any valid Applesoft expression). If a short machine language routine is stored beginning in location \$300 and your Applesoft program is short, there will be no problem in integrating the machine language program with the Applesoft program. If the Applesoft program is long, you can use HIMEM: expr. and then store the machine language program in memory locations above the value given by expr. Your machine language program will be safe in those locations.

## Debugging Programs

Debugging machine language programs is a difficult task, a few debugging aids are described next. On the standard Apple II, the Step and Trace functions are available (these features are not available on the Apple II Plus). The Step command displays each machine language instruction in assembly form, executes it, and then displays the contents of the registers. To use the Step command, type the address where you wish the Stepping function to begin, followed by the letter S. For example,

```
*300S
```

might produce

```
0300- A9 00          LDA #$00
      A=00   X=00   Y=02   P=30   S=F8
```

The results displayed under the assembly listing of the command represent the HEX contents of the accumulator (A), X register (X), Y register (Y), processor status register (P), and the stack pointer (S).

If you use the S command, you should make sure that the register contents are as you had expected after each machine language instruction was executed. To pinpoint your "bugs," find those places in the program where the registers do not contain the values you expect.

The first step instruction must contain an address to initiate the stepping process. Step instructions that come after the first may be of the form

```
*S
```

This command will execute consecutive instructions in your program. Each

**\*S**

instruction will execute the command following the one executed by the last Step command. The monitor program automatically will figure out the address of each instruction for you, after you give the address of the first one to be executed.

The Step command can be used alternately with other monitor commands. After executing a Step command, you can examine memory or alter memory using standard monitor commands and then continue stepping using the

**\*S**

command.

The Trace command works just like the Step command, except that you need not type T after each instruction is processed. Trace automatically will execute your entire machine language program, one instruction at a time, and display the register contents after executing each instruction. The Trace function is initiated by giving a starting address followed by the letter T as in

**\*300T**

Trace will terminate if a BRK instruction is encountered (BRK stands for BReaK, op code 00). The only other way to stop a Trace is to use the RESET key.

For those who have the Apple II Plus, debugging requires other techniques, since Step and Trace functions are not available. The first thing you should try is a manual trace of your program. Go through your listing line by line and do exactly what the instructions tell you to do (on paper, if necessary). You will be surprised at how many mistakes you can pinpoint in this way. In fact, relying too heavily on Step or Trace may not be a good idea. The manual trace forces you really to dig into your machine language program. There are a few things you should watch out for. First, did you select the correct addressing mode for each command? It is very easy to confuse

`LDA $01 (zero-page addressing mode)`

with

`LDA #$01 (immediate addressing mode)`

especially if you use the mini-assembler to enter your machine language program. Check your listing very carefully as you trace it to make sure all the addressing modes are correct.

If your program looks perfectly logical and runs, but does not give the expected results, check those places where built-in subroutines are being used. Are you assuming that a value stored in the accumulator, X register, or Y register before using the subroutine is still available after using it? This may be a poor assumption. Try storing the value in a regular



memory location before using the subroutine. Load the value from memory and store it in the appropriate register after using the subroutine.

Another source of trouble is the branch command. Make sure all branches are made to the logically correct places, and that all variables have been properly initialized. In Applesoft, the RUN command sets all variables equal to zero at the beginning of the program. The Go command in machine language does not do this.

If your machine language program is rather long, the task of debugging it by use of a manual trace can be rather formidable. It is suggested that you pinpoint key sections in your program. You can temporarily place a RTS command in your program after the first key section. When you run the program with this temporary RTS instruction inserted, you will be testing only this first key section. Examine the contents of memory where intermediate results are stored after running the program to make sure that they are as expected. If all went well, you can be fairly sure that this section of the program is functioning. After you test the first section of your program, do not forget to eliminate the temporary RTS command and return your program to its original state. You then go on to the next key section of your program, placing a temporary RTS command after it. Running this version of your program gives you a way of checking the second section of your program. After running this shortened version of your program, again check the results in all storage locations used to store intermediate results. Upon continuing in this manner, you will find the first section of your program that does not function as you had expected. You can then focus your full attention on this part. With a little practice, you will be able quickly to isolate the bugs in your program.

This concludes our discussion of Apple machine language. You have not been exposed to all the commands available to the Apple machine language programmer (although they are listed in Appendix B), but those covered in this book are the most important ones. With the knowledge you now have, you should be able easily to understand more advanced texts on the subject. You will also be able to figure things out for yourself by experimenting with the Apple. After writing your own machine language programs and becoming more comfortable with the language, you too will come to appreciate the power and beauty of machine language programming.

### Review Questions

1. The command JMP(\$810) is an \_\_\_\_\_ addressing mode command.
2. The command JMP(\$392) means "branch to the address stored in locations \$\_\_\_\_\_ and \$\_\_\_\_\_."
3. The monitor command

00:00 08 00 09 00 0A 00 0B

is executed. Then the assembly language commands

LDX #03

STA (01,X)

would store the accumulator's contents in location \$\_\_\_\_\_.

4. The monitor command

00:00 08 00 09 00 0A 00 0B

is executed. Then the assembly language commands

LDY #06  
STA (02), Y

would store the accumulator's contents in location \$\_\_\_\_\_.

5. In the postindexed indirect addressing mode, the \_\_\_\_\_ register is used as an index.
6. To store \$8D in memory location \$350 from Applesoft, use the command  
POKE \_\_\_\_\_, \_\_\_\_\_.
7. To examine the contents of memory location \$850 from Applesoft, use the command \_\_\_\_\_.
8. The Applesoft \_\_\_\_\_ statement is used to execute a machine language subroutine from an Applesoft program.
9. The most significant byte of the HEX equivalent of the decimal number 1234 is to be stored in memory location \$810 and the least significant byte is to be stored in location \$811. This can be done from Applesoft by using the commands

110 POKE \_\_\_\_\_, \_\_\_\_\_  
120 POKE \_\_\_\_\_, \_\_\_\_\_

#### TRUE or FALSE

10. Preindexed indirect addressing mode commands are all 3 bytes long.
11. The only command available in the indirect addressing mode is JMP.
12. The assembly language command

STA (03, Y)

is invalid.

13. Parentheses are required in the PEEK statement.
14. The RTS command always returns control to the monitor program.
15. The monitor command

300T

will execute your entire program one instruction at a time. After each instruction is executed, the contents of the registers will be displayed.



## Programming Problems

1. Write a machine language program that asks whether the users want to add, subtract, or multiply. After inputting a response (using a code you devise), have the user enter the two HEX numbers to be operated on and use an indirect addressing mode JMP command to branch to a section of the program that performs the arithmetic and prints the result.
2. Write an Applesoft program that utilizes a machine language sort algorithm to find the median of a list of scores. To find the median of a list of  $N$  scores:
  - (a) You sort the scores from lowest to highest.
  - (b) The median then is the "middle score" in this sorted list:
    - i) If  $N$  is odd, the median is score number  $\frac{N+1}{2}$  in the sorted list (here the first score in the list is labeled score 1).
    - ii) If  $N$  is even, then there is no middle score. The median is defined to be the average of score number  $\frac{N}{2}$  and score number  $\frac{N}{2} + 1$  in the sorted list.

Use Applesoft to allow the user to input the scores from the keyboard, keep track of the number of scores entered, and store the scores in consecutive memory locations using a POKE command. Be sure to include appropriate prompting messages so that the user does not have to guess what information to enter from the keyboard. Your program should then call a machine language subroutine that sorts the list of scores. When you return to Applesoft, use the sorted list to find the median and print it. Be sure to label the result clearly, producing output such as THE MEDIAN OF THE SCORES IS 82.

3. Input a four-digit address  $A$  from the keyboard. Print the HEX numbers stored in the 10 consecutive locations starting with location  $A$ . Use either preindexed indirect addressing or postindexed indirect addressing to read the values from memory.
4. Let  $X(I,J)$  stand for the element in row  $I$ , column  $J$ , of the matrix  $X$ . Let  $A$  and  $B$  be matrices with  $R$  rows and  $C$  columns (we say the *dimensions* of  $A$  and  $B$  are  $R \times C$ ). To find the sum  $S$  of matrices  $A$  and  $B$ , use this rule:

$$S(I,J) = A(I,J) + B(I,J) \quad \text{for each } I \text{ and } J \text{ such that} \\ 1 \leq I \leq R \\ 1 \leq J \leq C$$

In other words, to add two matrices, add corresponding elements together. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1+7 & 2+8 & 3+9 \\ 4+10 & 5+11 & 6+12 \end{bmatrix} = \begin{bmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

Notice that in order to add two matrices, each matrix must have the same dimensions. The dimensions of the sum will be the same as the dimensions of the matrices being added.

Store two  $4 \times 4$  matrices in memory as follows:

*Matrix A*

Row 1 begins in \$800  
Row 2 begins in \$808  
Row 3 begins in \$810  
Row 4 begins in \$818

*Matrix B*

Row 1 begins in \$820  
Row 2 begins in \$828  
Row 3 begins in \$830  
Row 4 begins in \$838

(Note that many locations between \$800 and \$840 will not contain any element of either matrix. For example, locations \$800 to \$803 will contain the elements in row 1 of matrix A, but locations \$804 to \$807 will not contain any elements from matrix A. Although this method of storing the elements from the matrices wastes storage space, it does make it easy to find the address in which the first element in each row is stored.)

Use the postindexed indirect addressing mode to find  $A + B$ . Note that the commands LDA, STA, and ADC are all available in the postindexed indirect addressing mode. You may store the address of the first element in each row of the matrices in memory using the monitor before you run your program.

5. Rewrite the matrix addition program of problem 4 using preindexed indirect addressing mode commands.



# ***Appendix A***

## ***Answers to Selected Review Questions***

### **Chapter 1**

1. (a) B2
2. (a) 00111101
3. (a) 23
4. (a) A9

### **Chapter 2**

1. 300 to 3FF
3. Op code, operands
5. Implied
7.  $1000 < 300.340M$
9. True
11. True
13. True
15. False

### **Chapter 3**

1. 88
3. Two

- 5. (b) GET
- 7. 46
- 9. \$4E8
- 11. False
- 13. True
- 15. False
- 17. True

**Chapter 4**

- 1. F4
- 3. BSAVE LOOP, A\$300, L\$71
- 5. Relative
- 7. D2
- 9. 0A
- 11. GOTO
- 13. JSR
- 15. True
- 17. True
- 19. True
- 21. True
- 23. False

**Chapter 5**

- 1. Exclamation point
- 3. BNE 31A
- 5. \$300L
- 7. 350, 351
- 9. False
- 11. False
- 13. True

**Chapter 6**

- 1. CLC
- 3. 1B
- 5. SEC
- 7. 04, 05, 0
- 9. True
- 11. False



- 13. False
- 15. False

## Chapter 7

- 1. Shape definitions, index
- 3. Zeros
- 5. 24 3C 27 2C 2D 2D 35 3E 37 36 06 00  
(Plotting vectors used for the shape were

! ! ! → → ! ! → → → → → →    ! ! → → ! ! ! !

- 7. 

|                   |                         |                                     |
|-------------------|-------------------------|-------------------------------------|
| 02 00 06 00 0E 00 | 36 3E 37 2E 35 36 06 00 | 24 3C 27 2C 2D 2D 35 3E 37 36 06 00 |
| ←Index→           | First shape definition  | Second shape definition             |

(Plotting vectors used for the first shape were

! ! ! → → ! ! → → ! ! ! !

- 9. 1
- 11. DRAW 3 would draw the third shape definition in a shape table starting at the last point plotted by the previous DRAW, XDRAW, or HPLOT command.
- 13. True
- 15. True
- 17. True
- 19. False
- 21. False
- 23. True

## Chapter 8

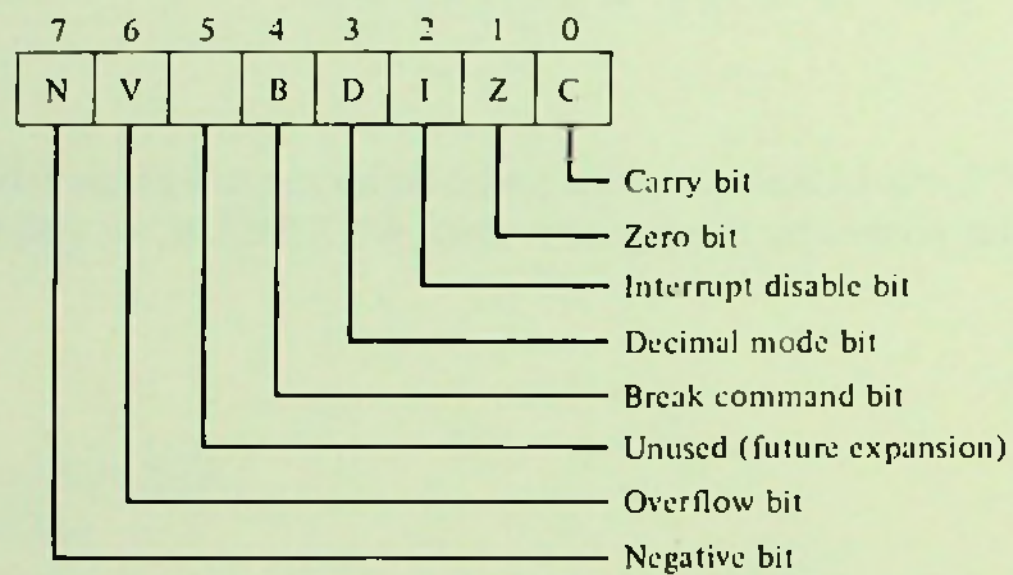
- 1. indirect
- 3. A00
- 5. Y
- 7. PRINT PEEK (2128)
- 9. 110 POKE 2064, 04  
120 POKE 2065, 210
- 11. True
- 13. True
- 15. True

# ***Appendix B***

## ***Op Codes***

### **Processor Status Register**

This diagram shows what each of the bits in the processor status register is used for.





## Addressing Modes

In the table of op codes that follows, these shorthand names are used to refer to the various addressing modes. The length of commands in each addressing mode is given here.

| <i>Shorthand Name</i> | <i>Addressing Mode</i>                                                  | <i>Length (Bytes)</i> |
|-----------------------|-------------------------------------------------------------------------|-----------------------|
| Immediate             | Immediate addressing mode                                               | 2                     |
| Zero Page             | Zero-page addressing mode                                               | 2                     |
| Accumulator           | Accumulator addressing mode                                             | 1                     |
| Zero page, X          | Zero-page indexed addressing mode,<br>with X register used as the index | 2                     |
| Absolute              | Absolute addressing mode                                                | 3                     |
| Absolute, X           | Absolute indexed addressing mode,<br>with X register used as the index  | 3                     |
| Absolute, Y           | Absolute indexed addressing mode,<br>with Y register used as the index  | 3                     |
| (Indirect, X)         | Preindexed indirect addressing mode                                     | 2                     |
| (Indirect), Y         | Postindexed indirect addressing mode                                    | 2                     |
| Indirect              | Indirect addressing mode                                                | 3                     |
| Implied               | Implied addressing mode                                                 | 1                     |
| Relative              | Relative addressing mode                                                | 2                     |
| Zero Page, Y          | Zero-page indexed addressing mode,<br>with Y register used as the index | 2                     |

In the assembly language forms given in the table of op codes, "Oper" stands for operand.

| <i>Mnemonic Code<br/>and Description</i> | <i>Addressing<br/>Mode</i> | <i>Op Codes</i>                   |                        |
|------------------------------------------|----------------------------|-----------------------------------|------------------------|
|                                          |                            | <i>Assembly<br/>Language Form</i> | <i>HEX<br/>Op Code</i> |
| ADC                                      |                            |                                   |                        |
| Add with Carry                           | Immediate                  | ADC #Oper                         | 69                     |
|                                          | Zero Page                  | ADC Oper                          | 65                     |
|                                          | Zero Page, X               | ADC Oper, X                       | 75                     |
|                                          | Absolute                   | ADC Oper                          | 6D                     |
|                                          | Absolute, X                | ADC Oper, X                       | 7D                     |
|                                          | Absolute, Y                | ADC Oper, Y                       | 79                     |
|                                          | (Indirect, X)              | ADC (Oper, X)                     | 61                     |
|                                          | (Indirect), Y              | ADC (Oper), Y                     | 71                     |

| Mnemonic Code<br>and Description         | Addressing<br>Mode | Op Codes                  |                |
|------------------------------------------|--------------------|---------------------------|----------------|
|                                          |                    | Assembly<br>Language Form | HEX<br>Op Code |
| AND                                      |                    |                           |                |
| "AND" with bits<br>of accumulator        | Immediate          | AND #Oper                 | 29             |
|                                          | Zero Page          | AND Oper                  | 25             |
|                                          | Zero Page, X       | AND Oper, X               | 35             |
|                                          | Absolute           | AND Oper                  | 2D             |
|                                          | Absolute, X        | AND Oper, X               | 3D             |
|                                          | Absolute, Y        | AND Oper, Y               | 39             |
|                                          | (Indirect, X)      | AND (Oper, X)             | 21             |
|                                          | (Indirect), Y      | AND (Oper), Y             | 31             |
| ASL                                      |                    |                           |                |
| Shift bits once<br>to left               | Accumulator        | ASL                       | 0A             |
|                                          | Zero Page          | ASL Oper                  | 06             |
|                                          | Zero Page, X       | ASL Oper, X               | 16             |
|                                          | Absolute           | ASL Oper                  | 0E             |
|                                          | Absolute, X        | ASL Oper, X               | 1E             |
| BCC                                      |                    |                           |                |
| Branch if Carry<br>bit is clear          | Relative           | BCC Oper                  | 90             |
| BCS                                      |                    |                           |                |
| Branch if Carry<br>bit is set = 1        | Relative           | BCS Oper                  | B0             |
| BEQ                                      |                    |                           |                |
| Branch if previous<br>result is zero     | Relative           | BEQ Oper                  | F0             |
| BIT                                      |                    |                           |                |
| Test bits with<br>accumulator            | Zero Page          | BIT Oper                  | 24             |
|                                          | Absolute           | BIT Oper                  | 2C             |
| BMI                                      |                    |                           |                |
| Branch if previous<br>result is minus    | Relative           | BMI Oper                  | 30             |
| BNE                                      |                    |                           |                |
| Branch if previous<br>result is not zero | Relative           | BNE Oper                  | D0             |
| BPL                                      |                    |                           |                |
| Branch if previous<br>result is plus     | Relative           | BPL Oper                  | 10             |



| Op Codes                                              |                                                                                                                    |                                                                                                                  |                                              |
|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| <i>Mnemonic Code<br/>and Description</i>              | <i>Addressing<br/>Mode</i>                                                                                         | <i>Assembly<br/>Language Form</i>                                                                                | <i>HEX<br/>Op Code</i>                       |
| BRK<br>Break                                          | Implied                                                                                                            | BRK                                                                                                              | 00                                           |
| BVC<br>Branch if overflow<br>bit is clear             | Relative                                                                                                           | BVC Oper                                                                                                         | 50                                           |
| BVS<br>Branch if overflow<br>bit is set = 1           | Relative                                                                                                           | BVS Oper                                                                                                         | 70                                           |
| CLC<br>Clear the Carry bit                            | Implied                                                                                                            | CLC                                                                                                              | 18                                           |
| CLD<br>Clear the<br>Decimal Mode                      | Implied                                                                                                            | CLD                                                                                                              | D8                                           |
| CLI<br>Clear the Interrupt<br>Disable status          | Implied                                                                                                            | CLI                                                                                                              | 58                                           |
| CLV<br>Clear the overflow bit                         | Implied                                                                                                            | CLV                                                                                                              | B8                                           |
| CMP<br>Compare with<br>contents of the<br>accumulator | Immediate<br>Zero Page<br>Zero Page, X<br>Absolute<br>Absolute, X<br>Absolute, Y<br>(Indirect, X)<br>(Indirect), Y | CMP #Oper<br>CMP Oper<br>CMP Oper, X<br>CMP Oper<br>CMP Oper, X<br>CMP Oper, Y<br>CMP (Oper, X)<br>CMP (Oper), Y | C9<br>C5<br>D5<br>CD<br>DD<br>D9<br>C1<br>D1 |
| CPX<br>Compare with<br>contents of the<br>X register  | Immediate<br>Zero Page<br>Absolute                                                                                 | CPX #Oper<br>CPX Oper<br>CPX Oper                                                                                | E0<br>E4<br>EC                               |

| Mnemonic Code<br>and Description              | Addressing<br>Mode | Op Codes                  |                |
|-----------------------------------------------|--------------------|---------------------------|----------------|
|                                               |                    | Assembly<br>Language Form | HEX<br>Op Code |
| CPY                                           |                    |                           |                |
| Compare with<br>contents of the<br>Y register | Immediate          | CPY #Oper                 | C0             |
|                                               | Zero Page          | CPY Oper                  | C4             |
|                                               | Absolute           | CPY Oper                  | CC             |
| DEC                                           |                    |                           |                |
| Decrement the<br>value in memory              | Zero Page          | DEC Oper                  | C6             |
|                                               | Zero Page, X       | DEC Oper, X               | D6             |
|                                               | Absolute           | DEC Oper                  | CE             |
|                                               | Absolute, X        | DEC Oper, X               | DE             |
| DEX                                           |                    |                           |                |
| Decrement the<br>value in the<br>X register   | Implied            | DEX                       | CA             |
| DEY                                           |                    |                           |                |
| Decrement the<br>value in the<br>Y register   | Implied            | DEY                       | 88             |
| EOR                                           |                    |                           |                |
| “Exclusive OR”<br>with bits of<br>accumulator | Immediate          | EOR #Oper                 | 49             |
|                                               | Zero Page          | EOR Oper                  | 45             |
|                                               | Zero Page, X       | EOR Oper, X               | 55             |
|                                               | Absolute           | EOR Oper                  | 4D             |
|                                               | Absolute, X        | EOR Oper, X               | 5D             |
|                                               | Absolute, Y        | EOR Oper, Y               | 59             |
|                                               | (Indirect, X)      | EOR (Oper, X)             | 41             |
|                                               | (Indirect), Y      | EOR (Oper), Y             | 51             |
| INC                                           |                    |                           |                |
| Increment the<br>value in memory              | Zero Page          | INC Oper                  | E6             |
|                                               | Zero Page, X       | INC Oper, X               | F6             |
|                                               | Absolute           | INC Oper                  | EE             |
|                                               | Absolute, X        | INC Oper, X               | FE             |



| Op Codes                                           |                                                                                                                    |                                                                                                                  |                                              |
|----------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| <i>Mnemonic Code<br/>and Description</i>           | <i>Addressing<br/>Mode</i>                                                                                         | <i>Assembly<br/>Language Form</i>                                                                                | <i>HEX<br/>Op Code</i>                       |
| INX<br>Increment the<br>value in the<br>X register | Implied                                                                                                            | INX                                                                                                              | E8                                           |
| INY<br>Increment the<br>value in the<br>Y register | Implied                                                                                                            | INY                                                                                                              | C8                                           |
| JMP<br>Unconditional jump                          | Absolute<br>Indirect                                                                                               | JMP Oper<br>JMP (Oper)                                                                                           | 4C<br>6C                                     |
| JSR<br>Jump to subroutine                          | Absolute                                                                                                           | JSR Oper                                                                                                         | 20                                           |
| LDA<br>Load the accumulator                        | Immediate<br>Zero Page<br>Zero Page, X<br>Absolute<br>Absolute, X<br>Absolute, Y<br>(Indirect, X)<br>(Indirect), Y | LDA #Oper<br>LDA Oper<br>LDA Oper, X<br>LDA Oper<br>LDA Oper, X<br>LDA Oper, Y<br>LDA (Oper, X)<br>LDA (Oper), Y | A9<br>A5<br>B5<br>AD<br>BD<br>B9<br>A1<br>B1 |
| LDX<br>Load the X register                         | Immediate<br>Zero Page<br>Zero Page, Y<br>Absolute<br>Absolute, Y                                                  | LDX #Oper<br>LDX Oper<br>LDX Oper, Y<br>LDX Oper<br>LDX Oper, Y                                                  | A2<br>A6<br>B6<br>AE<br>BE                   |
| LDY<br>Load the Y register                         | Immediate<br>Zero Page<br>Zero Page, X<br>Absolute<br>Absolute, X                                                  | LDY #Oper<br>LDY Oper<br>LDY Oper, X<br>LDY Oper<br>LDY Oper, X                                                  | A0<br>A4<br>B4<br>AC<br>BC                   |

| Op Codes                                                        |                    |                           |                |
|-----------------------------------------------------------------|--------------------|---------------------------|----------------|
| Mnemonic Code<br>and Description                                | Addressing<br>Mode | Assembly<br>Language Form | HEX<br>Op Code |
| LSR                                                             |                    |                           |                |
| Shift bits once<br>to the right                                 | Accumulator        | LSR                       | 4A             |
|                                                                 | Zero Page          | LSR Oper                  | 46             |
|                                                                 | Zero Page, X       | LSR Oper, X               | 56             |
|                                                                 | Absolute           | LSR Oper                  | 4E             |
|                                                                 | Absolute, X        | LSR Oper, X               | 5E             |
| NOP                                                             |                    |                           |                |
| No operation                                                    | Implied            | NOP                       | EA             |
| ORA                                                             |                    |                           |                |
| "OR" with bits<br>of accumulator                                | Immediate          | ORA #Oper                 | 09             |
|                                                                 | Zero Page          | ORA Oper                  | 05             |
|                                                                 | Zero Page, X       | ORA Oper, X               | 15             |
|                                                                 | Absolute           | ORA Oper                  | 0D             |
|                                                                 | Absolute, X        | ORA Oper, X               | 1D             |
|                                                                 | Absolute, Y        | ORA Oper, Y               | 19             |
|                                                                 | (Indirect, X)      | ORA (Oper, X)             | 01             |
|                                                                 | (Indirect), Y      | ORA (Oper), Y             | 11             |
| PHA                                                             |                    |                           |                |
| Push value in<br>accumulator onto<br>the stack                  | Implied            | PHA                       | 48             |
| PHP                                                             |                    |                           |                |
| Push value in<br>processor status<br>register onto<br>the stack | Implied            | PHP                       | 08             |
| PLA                                                             |                    |                           |                |
| Pull value off<br>stack and place<br>in accumulator             | Implied            | PLA                       | 68             |
| PLP                                                             |                    |                           |                |
| Pull value of<br>processor status<br>register off<br>the stack  | Implied            | PLP                       | 28             |



| Op Codes                         |                    |                           |                |
|----------------------------------|--------------------|---------------------------|----------------|
| Mnemonic Code<br>and Description | Addressing<br>Mode | Assembly<br>Language Form | HEX<br>Op Code |
| ROL                              |                    |                           |                |
| Rotate bits once<br>to left      | Accumulator        | ROL                       | 2A             |
|                                  | Zero Page          | ROL Oper                  | 26             |
|                                  | Zero Page, X       | ROL Oper, X               | 36             |
|                                  | Absolute           | ROL Oper                  | 2E             |
|                                  | Absolute, X        | ROL Oper, X               | 3E             |
| ROR                              |                    |                           |                |
| Rotate bits once<br>to right     | Accumulator        | ROR                       | 6A             |
|                                  | Zero Page          | ROR Oper                  | 66             |
|                                  | Zero Page, X       | ROR Oper, X               | 76             |
|                                  | Absolute           | ROR Oper                  | 6E             |
|                                  | Absolute, X        | ROR Oper, X               | 7E             |
| RTI                              |                    |                           |                |
| Return from interrupt            | Implied            | RTI                       | 40             |
| RTS                              |                    |                           |                |
| Return from<br>subroutine        | Implied            | RTS                       | 60             |
| SBC                              |                    |                           |                |
| Subtract with borrow             | Immediate          | SBC #Oper                 | E9             |
|                                  | Zero Page          | SBC Oper                  | E5             |
|                                  | Zero Page, X       | SBC Oper, X               | F5             |
|                                  | Absolute           | SBC Oper                  | ED             |
|                                  | Absolute, X        | SBC Oper, X               | FD             |
|                                  | Absolute, Y        | SBC Oper, Y               | F9             |
|                                  | (Indirect, X)      | SBC (Oper, X)             | E1             |
|                                  | (Indirect), Y      | SBC (Oper), Y             | F1             |
| SEC                              |                    |                           |                |
| Set the Carry bit = 1            | Implied            | SEC                       | 38             |
| SED                              |                    |                           |                |
| Set the Decimal Mode             | Implied            | SED                       | F8             |
| SEI                              |                    |                           |                |
| Set Interrupt<br>Disable status  | Implied            | SEI                       | 78             |

| Op Codes                                             |                    |                           |                |
|------------------------------------------------------|--------------------|---------------------------|----------------|
| Mnemonic Code<br>and Description                     | Addressing<br>Mode | Assembly<br>Language Form | HEX<br>Op Code |
| STA                                                  |                    |                           |                |
| Store the<br>accumulator's value                     | Zero Page          | STA Oper                  | 85             |
|                                                      | Zero Page, X       | STA Oper, X               | 95             |
|                                                      | Absolute           | STA Oper                  | 8D             |
|                                                      | Absolute, X        | STA Oper, X               | 9D             |
|                                                      | Absolute, Y        | STA Oper, Y               | 99             |
|                                                      | (Indirect, X)      | STA (Oper, X)             | 81             |
|                                                      | (Indirect), Y      | STA (Oper), Y             | 91             |
| STX                                                  |                    |                           |                |
| Store the<br>X register's value                      | Zero Page          | STX Oper                  | 86             |
|                                                      | Zero Page, Y       | STX Oper, Y               | 96             |
|                                                      | Absolute           | STX Oper                  | 8E             |
| STY                                                  |                    |                           |                |
| Store the<br>Y register's value                      | Zero Page          | STY Oper                  | 84             |
|                                                      | Zero Page, X       | STY Oper, X               | 94             |
|                                                      | Absolute           | STY Oper                  | 8C             |
| TAX                                                  |                    |                           |                |
| Transfer accumulator's value to the X register       | Implied            | TAX                       | AA             |
| TAY                                                  |                    |                           |                |
| Transfer accumulator's value to the Y register       | Implied            | TAY                       | A8             |
| TSX                                                  |                    |                           |                |
| Transfer the stack pointer's value to the X register | Implied            | TSX                       | BA             |
| TXA                                                  |                    |                           |                |
| Transfer X register's value to the accumulator       | Implied            | TXA                       | 8A             |
| TXS                                                  |                    |                           |                |
| Transfer the X register's value to the stack pointer | Implied            | TXS                       | 9A             |



| <i>Mnemonic Code<br/>and Description</i>                    | <i>Addressing<br/>Mode</i> | <i>Op Codes</i>                   |                        |
|-------------------------------------------------------------|----------------------------|-----------------------------------|------------------------|
|                                                             |                            | <i>Assembly<br/>Language Form</i> | <i>HEX<br/>Op Code</i> |
| TYA<br>Transfer Y register's<br>value to the<br>accumulator | Implied                    | TYA                               | 98                     |

The following table is convenient for converting HEX op codes into assembly language programs.

| <i>HEX<br/>Op Code</i> | <i>Mnemonic Code and<br/>Addressing Mode</i> |               | <i>HEX<br/>Op Code</i> | <i>Mnemonic Code and<br/>Addressing Mode</i> |               |
|------------------------|----------------------------------------------|---------------|------------------------|----------------------------------------------|---------------|
| 00                     | BRK                                          | Implied       | 35                     | AND                                          | Zero Page, X  |
| 01                     | ORA                                          | (Indirect, X) | 36                     | ROL                                          | Zero Page, X  |
| 05                     | ORA                                          | Zero Page     | 38                     | SEC                                          | Implied       |
| 06                     | ASL                                          | Zero Page     | 39                     | AND                                          | Absolute, Y   |
| 08                     | PHP                                          | Implied       | 3D                     | AND                                          | Absolute, X   |
| 09                     | ORA                                          | Immediate     | 3E                     | ROL                                          | Absolute, X   |
| 0A                     | ASL                                          | Accumulator   | 40                     | RTI                                          | Implied       |
| 0D                     | ORA                                          | Absolute      | 41                     | EOR                                          | (Indirect, X) |
| 0E                     | ASL                                          | Absolute      | 45                     | EOR                                          | Zero Page     |
| 10                     | BPL                                          | Relative      | 46                     | LSR                                          | Zero Page     |
| 11                     | ORA                                          | (Indirect), Y | 48                     | PHA                                          | Implied       |
| 15                     | ORA                                          | Zero Page, X  | 49                     | EOR                                          | Immediate     |
| 16                     | ASL                                          | Zero Page, X  | 4A                     | LSR                                          | Accumulator   |
| 18                     | CLC                                          | Implied       | 4C                     | JMP                                          | Absolute      |
| 19                     | ORA                                          | Absolute, Y   | 4D                     | EOR                                          | Absolute      |
| 1D                     | ORA                                          | Absolute, X   | 4E                     | LSR                                          | Absolute      |
| 1E                     | ASL                                          | Absolute, X   | 50                     | BVC                                          | Relative      |
| 20                     | JSR                                          | Absolute      | 51                     | EOR                                          | (Indirect), Y |
| 21                     | AND                                          | (Indirect, X) | 55                     | EOR                                          | Zero Page, X  |
| 24                     | BIT                                          | Zero Page     | 56                     | LSR                                          | Zero Page, X  |
| 25                     | AND                                          | Zero Page     | 58                     | CLI                                          | Implied       |
| 26                     | ROL                                          | Zero Page     | 59                     | EOR                                          | Absolute, Y   |
| 28                     | PLP                                          | Implied       | 5D                     | EOR                                          | Absolute, X   |
| 29                     | AND                                          | Immediate     | 5E                     | LSR                                          | Absolute, X   |
| 2A                     | ROL                                          | Accumulator   | 60                     | RTS                                          | Implied       |
| 2C                     | BIT                                          | Absolute      | 61                     | ADC                                          | (Indirect, X) |
| 2D                     | AND                                          | Absolute      | 65                     | ADC                                          | Zero Page     |
| 2E                     | ROL                                          | Absolute      | 66                     | ROR                                          | Zero Page     |
| 30                     | BMI                                          | Relative      | 68                     | PLA                                          | Implied       |
| 31                     | AND                                          | (Indirect), Y | 69                     | ADC                                          | Immediate     |

| <i>HEX<br/>Op Code</i> | <i>Mnemonic Code and<br/>Addressing Mode</i> | <i>HEX<br/>Op Code</i> | <i>Mnemonic Code and<br/>Addressing Mode</i> |
|------------------------|----------------------------------------------|------------------------|----------------------------------------------|
| 6A                     | ROR Accumulator                              | AE                     | LDX Absolute                                 |
| 6C                     | JMP Indirect                                 | B0                     | BCS Relative                                 |
| 6D                     | ADC Absolute                                 | B1                     | LDA (Indirect), Y                            |
| 6E                     | ROR Absolute                                 | B4                     | LDY Zero Page, X                             |
| 70                     | BVS Relative                                 | B5                     | LDA Zero Page, X                             |
| 71                     | ADC (Indirect), Y                            | B6                     | LDX Zero Page, Y                             |
| 75                     | ADC Zero Page, X                             | B8                     | CLV Implied                                  |
| 76                     | ROR Zero Page, X                             | B9                     | LDA Absolute, Y                              |
| 78                     | SEI Implied                                  | BA                     | TSX Implied                                  |
| 79                     | ADC Absolute, Y                              | BC                     | LDY Absolute, X                              |
| 7D                     | ADC Absolute, X                              | BD                     | LDA Absolute, X                              |
| 7E                     | ROR Absolute, X                              | BE                     | LDX Absolute, Y                              |
| 81                     | STA (Indirect, X)                            | C0                     | CPY Immediate                                |
| 84                     | STY Zero Page                                | C1                     | CMP (Indirect, X)                            |
| 85                     | STA Zero Page                                | C4                     | CPY Zero Page                                |
| 86                     | STX Zero Page                                | C5                     | CMP Zero Page                                |
| 88                     | DEY Implied                                  | C6                     | DEC Zero Page                                |
| 8A                     | TXA Implied                                  | C8                     | INY Implied                                  |
| 8C                     | STY Absolute                                 | C9                     | CMP Immediate                                |
| 8D                     | STA Absolute                                 | CA                     | DEX Implied                                  |
| 8E                     | STX Absolute                                 | CC                     | CPY Absolute                                 |
| 90                     | BCC Relative                                 | CD                     | CMP Absolute                                 |
| 91                     | STA (Indirect), Y                            | CE                     | DEC Absolute                                 |
| 94                     | STY Zero Page, X                             | D0                     | BNE Relative                                 |
| 95                     | STA Zero Page, X                             | D1                     | CMP (Indirect), Y                            |
| 96                     | STX Zero Page, Y                             | D5                     | CMP Zero Page, X                             |
| 98                     | TYA Implied                                  | D6                     | DEC Zero Page, X                             |
| 99                     | STA Absolute, Y                              | D8                     | CLD Implied                                  |
| 9A                     | TXS Implied                                  | D9                     | CMP Absolute, Y                              |
| 9D                     | STA Absolute, X                              | DD                     | CMP Absolute, X                              |
| A0                     | LDY Immediate                                | DE                     | DEC Absolute, X                              |
| A1                     | LDA (Indirect, X)                            | E0                     | CPX Immediate                                |
| A2                     | LDX Immediate                                | E1                     | SBC (Indirect, X)                            |
| A4                     | LDY Zero Page                                | E4                     | CPX Zero Page                                |
| A5                     | LDA Zero Page                                | E5                     | SBC Zero Page                                |
| A6                     | LDX Zero Page                                | E6                     | INC Zero Page                                |
| A8                     | TAY Implied                                  | E8                     | INX Implied                                  |
| A9                     | LDA Immediate                                | E9                     | SBC Immediate                                |
| AA                     | TAX Implied                                  | EA                     | NOP Implied                                  |
| AC                     | LDY Absolute                                 | EC                     | CPX Absolute                                 |
| AD                     | LDA Absolute                                 | ED                     | SBC Absolute                                 |



| <i>HEX<br/>Op Code</i> | <i>Mnemonic Code and<br/>Addressing Mode</i> | <i>HEX<br/>Op Code</i> | <i>Mnemonic Code and<br/>Addressing Mode</i> |
|------------------------|----------------------------------------------|------------------------|----------------------------------------------|
| EE                     | INC Absolute                                 | F0                     | BEQ Relative                                 |
|                        |                                              | F1                     | SBC (Indirect), Y                            |
|                        |                                              | F5                     | SBC Zero Page, X                             |
|                        |                                              | F6                     | INC Zero Page, X                             |
|                        |                                              | F8                     | SED Implied                                  |
|                        |                                              | F9                     | SBC Absolute, Y                              |
|                        |                                              | FD                     | SBC Absolute, X                              |
|                        |                                              | FE                     | INC Absolute, X                              |

## ***Appendix C***

# ***Low- and High-Resolution Colors***

In all cases, the actual color appearing on your screen will depend on color and tint adjustments.

### **Low-Resolution Graphics**

| <i>Decimal Value</i> | <i>HEX Value</i> | <i>Value to Store in \$30</i> | <i>Color</i> |
|----------------------|------------------|-------------------------------|--------------|
| 0                    | 0                | 00                            | Black        |
| 1                    | 1                | 11                            | Magenta      |
| 2                    | 2                | 22                            | Dark blue    |
| 3                    | 3                | 33                            | Purple       |
| 4                    | 4                | 44                            | Dark green   |
| 5                    | 5                | 55                            | Grey         |
| 6                    | 6                | 66                            | Medium blue  |
| 7                    | 7                | 77                            | Light blue   |
| 8                    | 8                | 88                            | Brown        |
| 9                    | 9                | 99                            | Orange       |
| 10                   | A                | AA                            | Grey         |
| 11                   | B                | BB                            | Pink         |
| 12                   | C                | CC                            | Green        |
| 13                   | D                | DD                            | Yellow       |
| 14                   | E                | EE                            | Aqua         |
| 15                   | F                | FF                            | White        |



**High-Resolution Graphics**

| <i>Decimal<br/>Value</i> | <i>Color</i> |
|--------------------------|--------------|
| 0                        | Black        |
| 1                        | Green        |
| 2                        | Violet       |
| 3                        | White        |
| 4                        | Black        |
| 5                        | Orange       |
| 6                        | Blue         |
| 7                        | White        |

## ***Appendix D***

# ***Summary of Useful Built-in Subroutines and CALL Locations***

To use these subroutines, jump to the address listed here. You can do this by using the JSR command if you are in machine language. If you are in Applesoft, use the CALL statement (or other BASIC equivalent). If a subroutine requires special data, make sure you store this information in the location mentioned in the Data Required by Subroutine section before attempting to use the subroutine. All registers scrambled by the subroutine (X register, Y register, or accumulator) are also listed here. It should be noted that this list contains some subroutines not discussed in the text.

| <i>HEX<br/>Address</i> | <i>BASIC<br/>Equivalent</i> | <i>Description<br/>of Subroutine</i>   | <i>Data Required<br/>by Subroutine</i>                                                                                 | <i>Registers<br/>Scrambled</i> |
|------------------------|-----------------------------|----------------------------------------|------------------------------------------------------------------------------------------------------------------------|--------------------------------|
| F800                   | PLOT C, R                   | Plots a low-resolution graphics point. | Accumulator:<br>Row R for point<br><br>Y register: Col-<br>umn C for<br>point<br><br>Location \$30:<br>Color for point | Accumulator                    |



| HEX<br>Address | BASIC<br>Equivalent | Description<br>of Subroutine                                                                                                                                                                                                                  | Data Required<br>by Subroutine                                                                                                                                   | Registers<br>Scrambled    |
|----------------|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------|
| F819           | HLIN A,B<br>AT C    | Plots a horizontal line in low-resolution graphics.                                                                                                                                                                                           | Location \$2C:<br>Ending column for line (B)<br>Y register: Starting column for line (A)<br>Accumulator:<br>Row for line (C)<br>Location \$30:<br>Color for line | Accumulator<br>Y register |
| F828           | VLIN A,B<br>AT C    | Plots a vertical line in low-resolution graphics.                                                                                                                                                                                             | Location \$2D:<br>Ending row for line (B)<br>Accumulator:<br>Starting row for line (A)<br>Y register: Column for line (C)<br>Location \$30:<br>Color for line    | Accumulator               |
| F940           | CALL -1728          | Prints the contents of the X and Y registers. The output will be a four-digit HEX number. The first pair of digits will give the contents of the Y register. The second pair of digits will give the contents of the X register.              |                                                                                                                                                                  | Accumulator               |
| F941           | CALL -1727          | Prints the contents of the accumulator and X register. The output will be a four-digit HEX number. The first pair of digits will give the contents of the accumulator and the second pair of digits will give the contents of the X register. |                                                                                                                                                                  | Accumulator               |

## 222 *Summary of Useful Built-in Subroutines and CALL Locations*

| <i>HEX<br/>Address</i> | <i>BASIC<br/>Equivalent</i> | <i>Description<br/>of Subroutine</i>                                                                                                                                       | <i>Data Required<br/>by Subroutine</i>   | <i>Registers<br/>Scrambled</i>                                                                      |
|------------------------|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------|-----------------------------------------------------------------------------------------------------|
| FB39                   | CALL -1223<br>or<br>TEXT    | Sets the TEXT mode.                                                                                                                                                        |                                          | Accumulator                                                                                         |
| FB40                   | CALL -1216<br>or<br>GR      | Sets the low-resolution graphics mode. Clears the screen of graphics if you are already in graphics mode.                                                                  |                                          | Accumulator<br>Y register.                                                                          |
| FC58                   | CALL -936<br>or<br>HOME     | Clears the screen in text mode.                                                                                                                                            |                                          | Accumulator<br>Y register                                                                           |
| FD1B                   | CALL -741                   | Reads the keyboard. Does not display a cursor prompt. The ASCII code for the key you strike will be stored in the accumulator. This subroutine operates like GET in BASIC. |                                          | Accumulator<br>filled with ASCII<br>code for the key<br>you strike                                  |
| FD35                   | CALL -715                   | This subroutine works like FD1B, except that escape codes will be interpreted by this subroutine and a cursor is displayed as a prompt.                                    |                                          | Accumulator<br>filled with ASCII<br>code for the key<br>you strike<br><br>Y register scram-<br>bled |
| FDDA                   | CALL -550                   | Prints the contents of the accumulator in HEX.                                                                                                                             | Load accumula-<br>tor with data          | Accumulator                                                                                         |
| FDED                   | CALL -531                   | Prints the character whose ASCII code is stored in the accumulator.                                                                                                        | Load accumula-<br>tor with ASCII<br>code | None                                                                                                |
| FE80                   | CALL -384<br>or<br>INVERSE  | Sets the INVERSE text mode. All characters (except those you type from the keyboard) will be printed black on a white back-<br>ground.                                     |                                          | Y register                                                                                          |



| <i>HEX<br/>Address</i> | <i>BASIC<br/>Equivalent</i>  | <i>Description<br/>of Subroutine</i>                                                   | <i>Data Required<br/>by Subroutine</i> | <i>Registers<br/>Scrambled</i> |
|------------------------|------------------------------|----------------------------------------------------------------------------------------|----------------------------------------|--------------------------------|
| FE84                   | CALL -380<br>or<br>NORMAL    | Sets the NORMAL text mode. All characters will be printed white on a black background. |                                        | Y register                     |
| FF3A                   | CALL -198<br>or<br>Control-G | Beeps Apple's speaker.                                                                 |                                        | Accumulator                    |
| FF69                   | CALL -151                    | Enters the Monitor program.                                                            |                                        | None                           |

# Appendix E

## ASCII Codes

A complete list of ASCII codes is given here for reference. The Control characters have special functions, but nothing will be printed on the screen if either JSR FDED (in machine language) or CHR\$ (in Applesoft) is used to "print" one of the characters.

### Control Characters

| Character     | HEX Code | Decimal Code |
|---------------|----------|--------------|
| Control A     | 81       | 129          |
| Control B     | 82       | 130          |
| Control C     | 83       | 131          |
| Control D     | 84       | 132          |
| Control E     | 85       | 133          |
| Control F     | 86       | 134          |
| Control G     | 87       | 135          |
| (←) Control H | 88       | 136          |
| Control I     | 89       | 137          |
| Control J     | 8A       | 138          |
| Control K     | 8B       | 139          |
| Control L     | 8C       | 140          |
| Control M*    | 8D       | 141          |
| Control N     | 8E       | 142          |
| Control O     | 8F       | 143          |
| Control P     | 90       | 144          |
| Control Q     | 91       | 145          |
| Control R     | 92       | 146          |
| Control S     | 93       | 147          |

\*Note that Control M causes a carriage return to be executed.



| Character       | HEX<br>Code | Decimal<br>Code |
|-----------------|-------------|-----------------|
| Control T       | 94          | 148 20          |
| (→) Control U   | 95          | 149             |
| Control V       | 96          | 150 22          |
| Control W       | 97          | 151             |
| Control X       | 98          | 152 24          |
| Control Y       | 99          | 153             |
| Control Z       | 9A          | 154 26          |
| ESC             | 9B          | 155             |
| Control-Shift M | 9D          | 157 28          |
| Control ^       | 9E          | 158             |

Bizimom@aol.com

## ASCII Codes

In machine language, you can use the subroutine FDED to print any character you wish. Load the accumulator with the appropriate HEX ASCII code and jump (JSR) to location FDED. You can print in NORMAL, INVERSE, or FLASH modes by selecting the proper ASCII code. Alternately you can use the subroutine FE80 to set the INVERSE mode and the subroutine FE84 to set the NORMAL mode in machine language.

In Applesoft, CHR\$ plays the role of the subroutine FDED (when CHR\$ is used in a PRINT statement). You can print a character by using CHR\$( ), where the decimal equivalent of the ASCII code for the character goes inside the parentheses of the CHR\$ function. The argument of the CHR\$ function must be between 0 and 255 (inclusive). Within this range, there is more than one ASCII code corresponding to each character, but only one of the values is listed here. Note that you *cannot* print INVERSE or FLASHing characters using the CHR\$ function. For this reason, the decimal equivalents of the hexadecimal INVERSE and FLASH ASCII codes are not given here. If you need to print INVERSE or FLASH characters in Applesoft, then set the mode to be printed using the INVERSE or FLASH statement.

| Character | NORMAL      |                 | INVERSE     | FLASH       |
|-----------|-------------|-----------------|-------------|-------------|
|           | HEX<br>Code | Decimal<br>Code | HEX<br>Code | HEX<br>Code |
| Space     | A0          | 160 32          | 20          | 60          |
| !         | A1          | 161             | 21          | 61          |
| "         | A2          | 162 34          | 22          | 62          |
| #         | A3          | 163             | 23          | 63          |
| \$        | A4          | 164 36          | 24          | 64          |
| %         | A5          | 165             | 25          | 65          |
| &         | A6          | 166 38          | 26          | 66          |
| '         | A7          | 167             | 27          | 67          |

CHR\$( )  
HEX 21 = Dec 33

| Character | NORMAL      |                 | INVERSE     | FLASH       |
|-----------|-------------|-----------------|-------------|-------------|
|           | HEX<br>Code | Decimal<br>Code | HEX<br>Code | HEX<br>Code |
| (         | A8          | 168             | 28          | 68          |
| )         | A9          | 169             | 29          | 69          |
| *         | AA          | 170             | 2A          | 6A          |
| +         | AB          | 171             | 2B          | 6B          |
| ,         | AC          | 172             | 2C          | 6C          |
| -         | AD          | 173             | 2D          | 6D          |
| .         | AE          | 174             | 2E          | 6E          |
| /         | AF          | 175             | 2F          | 6F          |
| 0         | B0          | 176             | 30          | 70          |
| 1         | B1          | 177             | 31          | 71          |
| 2         | B2          | 178             | 32          | 72          |
| 3         | B3          | 179             | 33          | 73          |
| 4         | B4          | 180             | 34          | 74          |
| 5         | B5          | 181             | 35          | 75          |
| 6         | B6          | 182             | 36          | 76          |
| 7         | B7          | 183             | 37          | 77          |
| 8         | B8          | 184             | 38          | 78          |
| 9         | B9          | 185             | 39          | 79          |
| :         | BA          | 186             | 3A          | 7A          |
| ;         | BB          | 187             | 3B          | 7B          |
| <         | BC          | 188             | 3C          | 7C          |
| =         | BD          | 189             | 3D          | 7D          |
| >         | BE          | 190             | 3E          | 7E          |
| ?         | BF          | 191             | 3F          | 7F          |
| @         | C0          | 192             | 00          | 40          |
| A         | C1          | 193             | 01          | 41          |
| B         | C2          | 194             | 02          | 42          |
| C         | C3          | 195             | 03          | 43          |
| D         | C4          | 196             | 04          | 44          |
| E         | C5          | 197             | 05          | 45          |
| F         | C6          | 198             | 06          | 46          |
| G         | C7          | 199             | 07          | 47          |
| H         | C8          | 200             | 08          | 48          |
| I         | C9          | 201             | 09          | 49          |
| J         | CA          | 202             | 0A          | 4A          |
| K         | CB          | 203             | 0B          | 4B          |
| L         | CC          | 204             | 0C          | 4C          |
| M         | CD          | 205             | 0D          | 4D          |
| N         | CE          | 206             | 0E          | 4E          |
| O         | CF          | 207             | 0F          | 4F          |
| P         | D0          | 208             | 10          | 50          |



209  
-123 (51)

| Character | NORMAL   |              | INVERSE  | FLASH    |
|-----------|----------|--------------|----------|----------|
|           | HEX Code | Decimal Code | HEX Code | HEX Code |
| Q         | D1       | 209 81       | 11       | 51       |
| R         | D2       | 210 82       | 12       | 52       |
| S         | D3       | 211 83       | 13       | 53       |
| T         | D4       | 212 84       | 14       | 54       |
| U         | D5       | 213 85       | 15       | 55       |
| V         | D6       | 214 86       | 16       | 56       |
| W         | D7       | 215 87       | 17       | 57       |
| X         | D8       | 216 88       | 18       | 58       |
| Y         | D9       | 217 89       | 19       | 59       |
| Z         | DA       | 218 90       | 1A       | 5A       |
| [         | DB       | 219 91       | 1B       | 5B       |
| \         | DC       | 220 92       | 1C       | 5C       |
| ]         | DD       | 221 93       | 1D       | 5D       |
| ^         | DE       | 222 94       | 1E       | 5E       |
| _         | DF       | 223 95       | 1F       | 5F       |

The [ character and the \ character cannot be accessed from Apple's keyboard. The underscore character is not accessible from the keyboard, either. The ] character is accessed by typing Shift-M.

CHR\$(33) = ! = HEX

CHR\$(1) =

# Appendix F

## Text Screen Map

If the ASCII code for any character is stored in text screen memory, then the character will be displayed. To find the memory location corresponding to any square on the text screen, use the rule

$$\text{Memory location for row R, column C} = \\ (\text{Memory location for row R, column 0}) + C$$

On the left side of the text screen map, the address corresponding to column 0 is given for each row. All numerals given on the map are in HEX. The same numbering system used throughout this book is shown on the text map. Rows and columns are both numbered beginning with zero. Thus the top row is labeled 0 and the left column is labeled 00.

For your convenience, the decimal address corresponding to column 00 in each row is given here.

| <i>HEX Row<br/>Number</i> | <i>HEX<br/>Address of<br/>Column 00</i> | <i>Decimal<br/>Address of<br/>Column 00</i> | <i>HEX Row<br/>Number</i> | <i>HEX<br/>Address of<br/>Column 00</i> | <i>Decimal<br/>Address of<br/>Column 00</i> |
|---------------------------|-----------------------------------------|---------------------------------------------|---------------------------|-----------------------------------------|---------------------------------------------|
| 00                        | \$400                                   | 1024                                        | 0C                        | \$628                                   | 1576                                        |
| 01                        | \$480                                   | 1152                                        | 0D                        | \$6A8                                   | 1704                                        |
| 02                        | \$500                                   | 1280                                        | 0E                        | \$728                                   | 1832                                        |
| 03                        | \$580                                   | 1408                                        | 0F                        | \$7A8                                   | 1960                                        |
| 04                        | \$600                                   | 1536                                        | 10                        | \$450                                   | 1104                                        |
| 05                        | \$680                                   | 1664                                        | 11                        | \$4D0                                   | 1232                                        |
| 06                        | \$700                                   | 1792                                        | 12                        | \$550                                   | 1360                                        |
| 07                        | \$780                                   | 1920                                        | 13                        | \$5D0                                   | 1488                                        |
| 08                        | \$428                                   | 1064                                        | 14                        | \$650                                   | 1616                                        |
| 09                        | \$4A8                                   | 1192                                        | 15                        | \$6D0                                   | 1744                                        |
| 0A                        | \$528                                   | 1320                                        | 16                        | \$750                                   | 1872                                        |
| 0B                        | \$5A8                                   | 1448                                        | 17                        | \$7D0                                   | 2000                                        |



|          | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |   |  |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|--|
| 0- S400  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 1- S480  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 2- S500  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 3- S580  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 4- S600  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 5- S680  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 6- S700  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 7- S780  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 8- S428  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 9- S4A8  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 1  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4 |  |
| A- S528  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 2  | 5  | 5  | 5  | 5  | 5  | 5  | 5  | 5  | 5  | 5  | 5  | 5  | 5 |  |
| B- S5A8  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| C- S628  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| D- S6A8  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| E- S728  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| F- S7A8  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 10- S450 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 11- S4D0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 12- S550 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 13- S5D0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 14- S650 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 15- S6D0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 16- S750 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |
| 17- S7D0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |  |

Text Screen Map  
(All numerals are HEX)

Row 12 (C28)  
Col 19 (13)  
638

# **Appendix G**

## **Glossary**

Page numbers in parentheses after each entry refer to the page where the term being defined was first introduced in the text.

**Absolute addressing mode (p. 15)**

Mode in which the address operated on by a command is given by the command's operand.

**Absolute indexed addressing mode (p. 25)**

Mode in which the address operated on by a command is found by adding the contents of the X (or Y) register to a base address. The base address is given by the command's operand.

**Accumulator (p. 10)**

Special location that is used to transfer data from one location to another; also, the location where arithmetic operations take place.

**Addressing mode (p. 13)**

One of the ways in which a machine language command can be used. The *addressing mode* of a command determines how the information contained in the command's operand is used. For example, if LDA is used in the immediate *addressing mode*, its operand is the number that gets loaded into the accumulator. If LDA is used in the absolute *addressing mode*, the operand is interpreted as the location in which the number to be loaded into the accumulator is stored.

**Algorithm (p. 77)**

A method used to solve a problem.

**AND (p. 37)**

A logical operator. If A and B stand for two statements that are either true or false, then the expression "A AND B" is true only when the statements A and B are both true. The expression is false if at least one of the two statements is false.

**Array (p. 183)**

A matrix. A set whose elements are referred to using two subscripts—one identifies the element's row and the other the element's column.

**ASCII (p. 32)**

An abbreviation for American Standard Code for Information Interchange. In this code, each character is represented by a HEX number. (See Appendix E.)

**Assembler (p. 11)**

A program that converts the three-letter mnemonic labels of assembly language to the



corresponding hexadecimal numerical codes of machine language. The process of converting the mnemonic labels of assembly language to machine language op codes and operands is called *assembling* the machine language program.

**Assembly language (p. 11)**

A language similar to machine language, except that commands are not coded numerically as in machine language. In *assembly language*, three-letter abbreviations (called mnemonics) are used to refer to the commands.

**Binary (p. 1)**

A number system whose two digits are 0 and 1. In this number system, the place value of each digit is two times the place value of the digit to the immediate right. *Binary* is also known as base 2.

**Bit (p. 4)**

A *Binary digit*. One of the individual digits in a binary number is called a *bit*.

**Bootting DOS (p. 55)**

The process of adding commands dealing with disk operation to the list of commands that are available in Applesoft. You must *boot DOS* (DOS stands for Disk Operating System) before you can access the DOS in order to store programs on the disk or retrieve programs from the disk.

**Branch (p. 50)**

A transfer of control from one part of a program to another part of the program

**Bubble sort (p. 77)**

A method used to put a list of numbers in either ascending or descending order. This method is based on comparing elements that are next to one another on the list to be sorted.

**Byte (p. 4)**

A string of 8 bits. Each one of Apple's memory locations can store 1 *byte*.

**C bit**

*See* Carry bit.

**Carry bit (p. 122)**

One of the 8 bits that is stored in a special memory location called the processor status register. This bit is used to keep track of "carrying" in addition and "borrowing" in subtraction.

**Debug (p. 87)**

To eliminate errors in a program.

**Decimal (p. 1)**

A number system whose 10 digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In this system, the place value of each digit is 10 times the place value of the digit to the immediate right. *Decimal* is also known as base 10.

**Decimal mode (p. 127)**

When this option is invoked, arithmetic is done using decimal arithmetic facts rather than hexadecimal arithmetic facts.

**Decrement (p. 68)**

To decrease by one.

**Disassemble (p. 16)**

To convert the numerical codes of machine language into the corresponding three-letter abbreviations (mnemonics) that are used to refer to commands in assembly language.

**Expand a numeral (p. 2)**

A process used to convert a number to base 10. To expand a number: (a) multiply each digit in the number by its place value; (b) add all the products found in step (a); the sum found in step (b) is the base 10 equivalent of the number.

**HEX**

See Hexadecimal.

**Hexadecimal (p. 1)**

A number system whose sixteen digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In this number system, the place value of each digit is 16 times the place value of the digit to the immediate right. *Hexadecimal* is also known as base sixteen.

**Immediate addressing mode (p. 15)**

Mode in which the datum processed by a command is given by the command's operand.

**Implied addressing mode (p. 15)**

Mode in which no operands are associated with a command because (a) the address operated on by the command is given explicitly by the command's op code or (b) all information needed to execute the command is contained in the command's op code.

**Increment (p. 67)**

To increase by one.

**Index in absolute indexed addressing (p. 46)**

A register (either the X or Y register) that stores a number that is used to select an item of interest from a list of items that is stored in consecutive memory locations.

**Index of a shape table (p. 153)**

Where the location of each shape definition in the shape table is stored. The total number of shape definitions in the table is also stored in the *index*. The *index* is the first part of the shape table.

**Indexed indirect addressing mode (p. 169)**

Mode in which the address operated on by a command is stored in two consecutive zero-page locations. These zero-page locations are pointed to by the sum of the command's operand and the contents of the X register. *Indexed indirect addressing mode* is also known as the preindexed indirect addressing mode.

**Indirect addressing mode (p. 175)**

Mode in which the address operated on by a command is stored in a location given by the command's operand. Strictly speaking, JMP is the only command available in the *indirect addressing mode*. However, the term *indirect addressing* is sometimes used in a generic sense to include the *indirect addressing mode*, the indexed indirect addressing mode, and the indirect indexed addressing mode.

**Interchange sort algorithm (p. 100)**

A method used to arrange a list of numbers in ascending or descending order. The method is based on comparing each number on the list to every other number on the list.

**Jump (p. 73)**

An unconditional branch.

**Least significant byte (LSB) (p. 11)**

The two rightmost digits in a four-digit address. In the address \$1234, 34 is the *LSB*. It takes



two memory locations to store a four-digit address. When an address is an operand, the *LSB* of the address is placed in the lower of the two locations.

**LSB**

See Least significant byte.

**Machine language (p. 1)**

A language in which all commands are referred to by hexadecimal numbers.

**Matrix**

See array.

**Median (p. 201)**

If a list of numbers is in ascending or descending order, then the *median* is the number in the middle of the list. Half the numbers on the list are larger than the *median* and half are smaller.

**Memory dump (p. 8)**

A display that shows the hexadecimal bytes that are stored in memory. Data displayed in a *dump* are “raw data”—no labels are supplied by the computer (as in an assembly listing) to help the programmer interpret the HEX bytes.

**Memory range (p. 8)**

A set of consecutive memory locations.

**Mini-assembler (p. 11)**

A program used to convert the three-letter abbreviations (mnemonics) of assembly language into the hexadecimal codes of machine language. The *mini-assembler* lacks the editing capabilities of an assembler.

**Mnemonic (p. 11)**

A memory aid. In assembly language, each command is given a three-letter abbreviation (*mnemonic label*). This is done because the corresponding op codes of machine language are very difficult to memorize. You can write your programs using the *mnemonic* labels of assembly language and use the mini-assembler to convert these *mnemonics* into the hexadecimal op codes of machine language.

**Monitor (p. 4)**

A program that allows you to examine the contents of any of Apple's memory locations. You can also use the *monitor* program to change the contents of any Random Access Memory (RAM) location. The *monitor* program has a command that is used to move blocks of data in memory. This command is used to edit machine language programs.

**Most significant byte (MSB) (p.11)**

The two leftmost digits in a four-digit address. In the address \$1234, 12 is the *MSB*. It takes two memory locations to store a four-digit address. When an address is an operand, the *MSB* of the address is placed in the higher of the two locations.

**MSB**

See Most significant byte.

**N bit**

See Negative bit.

**Negative bit (p. 56)**

One of the 8 bits that is stored in a special memory location called the processor status

register. The *negative bit* (*N-bit*) is set equal to 1 when an arithmetic command gives a negative result or when a negative number is moved (using a command such as LDA or STX).

**Negative number (p. 51)**

A number whose leftmost bit is one (these are the HEX numbers \$80 to \$FF inclusive).

**Nibble (p. 4)**

Four bits. Each digit in a two-digit HEX number represents a *nibble*.

**Op code (p. 11)**

OPERation code. In machine language, each command is given a two-digit hexadecimal code called an *op code*.

**Operand (p. 11)**

Data that the computer needs to carry out a command is given in the command's *operand*.

| <i>Machine Language</i> |                |    | <i>Assembly Language</i> |                |
|-------------------------|----------------|----|--------------------------|----------------|
| AD                      | 90             | 03 | LDA                      | 390            |
| op code                 | <i>operand</i> |    | mnemonic<br>code         | <i>operand</i> |

**P register**

See Processor status register.

**Page (p. 27)**

256 consecutive memory locations. The *page* number of a four-digit address is given by the two leftmost digits (the most significant byte). For example, the memory location \$0934 is in *page* 09 of memory.

**Plotting vector (p. 153)**

A symbol used to represent one of the moves that is made when tracing out a high-resolution shape.

**Plus (p. 52)**

A number is *plus* if its leftmost bit is zero. The numbers \$00 to \$7F inclusive are *plus* numbers. *Plus* numbers are also called positive numbers.

**Positive number**

See Plus.

**Postindexed indirect addressing mode**

See Indirect indexed addressing mode.

**Preindexed indirect addressing mode**

See Indexed indirect addressing mode.

**Processor status register (p. 56)**

A special memory location whose bits (a) keep track of "carrying" in addition and "borrowing" in subtraction, and (b) keep track of whether the result of a calculation is positive, negative, or zero. This information is used to determine program flow when a conditional branch command is executed.

**RAM**

See Random access memory.

**Random access memory (RAM) (p. 6)**

Memory whose contents can be both examined and altered by the programmer.



**Read only memory (ROM) (p. 6)**

Memory whose contents can be examined but not altered by the programmer. The information stored in *read only memory* is built into the Apple and is vital to its functioning.

**Register (p. 25)**

A special memory location referred to by name rather than by a number. These locations are used to perform special tasks in machine language programming. The *registers* used in machine language are the X register, Y register, accumulator (sometimes called the A register), and the processor status register (P register).

**Relative addressing mode (p. 60)**

The addressing mode used by conditional branch commands. The destination of a conditional jump is not specified by giving the address of the destination. Instead the operand of a *relative addressing mode* command tells Apple how far to move (forward or backward) from the command's address to reach the address of the destination.

**ROM**

See Read only memory.

**Shape definition (p. 153)**

The hexadecimal codes that represent the moves used to trace a high-resolution shape.

**Shape table (p. 153)**

A table containing the hexadecimal codes for one or more shape definitions. The table consists of two parts—an index (which gives the location of each shape definition in the table) and the shape definitions themselves. *Shape tables* are used together with BASIC commands to manipulate high-resolution shapes.

**X register (p. 25)**

A special memory location that is used as a counter. It is also used in absolute indexed addressing mode to select an element from a list of elements that is stored in consecutive memory locations.

**Y register (p. 25)**

A special memory location that is used just as the X register (see X register).

**Z bit**

See Zero bit.

**Zero bit (p. 56)**

One of the 8 bits that is stored in a special memory location called the processor status register. The *Zero bit* (*Z bit*) is set equal to one when an arithmetic command gives a zero result or when the number zero is moved using a command like LDA or STX.

**Zero-page addressing mode (p. 26)**

In this addressing mode, the zero-page address operated on by a command is given by the command's operand.

**Zero-page indexed addressing mode (p. 179)**

Addressing mode in which the zero-page address operated on by a command is found by adding the contents of the X (or Y) register to a base address. The base address is given by the command's operand. (Note: The Y register can be used as an index in *zero-page indexed addressing mode* only with the commands LDX and STX.)

**Zero-page memory (p. 27)**

The 256 memory locations in page zero of memory (locations \$0000 to \$00FF inclusive).



The first of these is the fact that the  
the second is the fact that the  
the third is the fact that the  
the fourth is the fact that the  
the fifth is the fact that the  
the sixth is the fact that the  
the seventh is the fact that the  
the eighth is the fact that the  
the ninth is the fact that the  
the tenth is the fact that the

The first of these is the fact that the  
the second is the fact that the  
the third is the fact that the  
the fourth is the fact that the  
the fifth is the fact that the  
the sixth is the fact that the  
the seventh is the fact that the  
the eighth is the fact that the  
the ninth is the fact that the  
the tenth is the fact that the

The first of these is the fact that the  
the second is the fact that the  
the third is the fact that the  
the fourth is the fact that the  
the fifth is the fact that the  
the sixth is the fact that the  
the seventh is the fact that the  
the eighth is the fact that the  
the ninth is the fact that the  
the tenth is the fact that the



# INDEX

- Absolute addressing mode, 15
  - op codes, 206-217
- Absolute indexed addressing mode, 45-46
  - storing information in consecutive memory locations, 46
  - subscripted variables 45-46
- Accumulator, 10
  - comparing data in memory, 59
  - effect of built-in subroutines on, 31-32, 35, 220-223
  - moving data, 10-11
- ADC 121-122, 126
  - decimal mode, 127-129
  - input of two digit numbers from keyboard, 129-134
- Addition
  - input of two digit numbers from keyboard, 129-134
  - of 2 two-digit numbers, 121-123
  - of 2 four-digit numbers, 124-127
  - using the monitor, 50
- Addressing modes, 207
  - notation for use with mini-assembler, 94, 177, 181, 183
  - see also* Absolute addressing mode, Absolute indexed addressing mode, Immediate addressing mode, Implied addressing mode, Indexed indirect addressing mode, Indirect addressing mode, Indirect indexed addressing mode, Postindexed indirect addressing mode, Preindexed indirect addressing mode, Relative addressing mode, Zero page addressing mode, Zero page indexed addressing mode
- Altering memory, 9-10
- AND, 37-39
  - absolute indexed addressing mode, 46
  - conversion of ASCII codes to numerical values, 39-40
  - NORMAL, INVERSE and FLASH ASCII Codes, 41-44, 225-227
- Arithmetic
  - addition of 2 two-digit numbers, 121-123
  - addition of 2 four-digit numbers, 124-127
  - addition using the monitor, 50
  - decimal arithmetic, 127-129
  - multiplication, 140-150
  - subtraction program, 138-140
  - subtraction using the monitor, 51
- Arrays (matrices)
  - machine language representation, 183
  - programming examples, 183-190
- "Art," computer-generated, 171
- ASCII Codes, 32
  - list of codes, 224-227
  - NORMAL, INVERSE and FLASH codes, 41-44, 225-227
- ASL, 130-131, 146
  - use with ROL, 146-148
- Assembling a machine language program, 11, 93
- Assembly language, 11
  - listings, 16-17
  - listings for relative addressing mode commands, 64-65
- BCC, 122
- BCS, 122
- BEQ, 59
- Binary (base 2,) 1
  - binary to HEX conversions, 3
  - HEX to binary conversions, 3
- Bits, 4
  - operation on bits using AND, 37-38



- BLOAD, 55
- BMI, 59
- BNE, 59
- Booting DOS without losing program, 55
- BPL, 59, 69
- Branch commands, 56, 59–65
  - operands, 52, 59–65
  - use with compare commands, 65–73
- BRK, 198
- BRUN, 55
- BSAVE, 54–55
- Bubble sort, 77–84
- Built-in subroutines, 25, 220–223
  - clear the screen, to, 47
  - effect on registers of, 31–32, 137, 220–223
  - finding addresses of, 28–29
  - input from keyboard, to, 35–36
  - plot a horizontal line, to, 29–30
  - plot a point, to, 26–27
  - plot a vertical line, to, 30–31
  - print a character, to, 33–34
  - print numerical information, to, 34–35
  - random numbers, to generate, 36–37
  - set graphics mode, to, 27
  - set text mode, to, 29
- Byte, 4
  
- C bit (processor status register), 122, 126
  - effect of ASL on, 130
  - effect of branch commands on, 128, 137
  - effect of ROL on, 146–148
  - effect of ROR on, 148–149
  - subtraction, 138–140
- CALL, 192–193, 220–223
  - machine language equivalent, of, 29
- Carriage return, 33
- Carry bit (processor status register). *See* C bit
- Changing contents of memory, 9–10
- CLC, 121–122
- CLD, 129
- CMP, 57–58
  - absolute addressing mode, 58
  - absolute indexed addressing mode, 58
  - immediate addressing mode, 57
- Colors
  - low and high resolution, list of, 218–219
  - storage of values, 26
- Compare command, 57, 58, 59
- Combinations, branch command, 65–73
- Converting from one base to another
  - binary to decimal, 1–2
  - binary to HEX, 3
  - decimal to HEX, 5 (problem), 193–195
  - HEX to binary, 3–4
  - HEX to decimal, 4 (problem), 191, 195
- CPX, 57
  - immediate addressing mode, 58
- CPY, 57, 58
- Cursor control, HOME, 47
  
- Debugging, 87, 137, 197–199
- DEC, 68
  - operation in decimal mode, 129
- Decimal mode, 127–129
- Decimal number system (base 10), 1
- DEX, 68
  - operation in decimal mode, 129
- DEY, 68
  - operation in decimal mode, 129
- Disassembling a machine language program, 16
- Disk
  - saving programs, 54–55
  - saving shape tables, 163
- DOS, booting, 55
- DRAW, 163, 165
  
- Editing machine language programs, 17–22
  - replacing a command with a longer one, 18–21
  - replacing a command with a shorter one, 21–22
  - replacing a command with one of the same length, 17
- Effect of built-in subroutines on registers, 31–32, 137, 220–223
- Entering a shape table, 162–163
- Examining memory with the monitor
  - memory ranges, 8–9
  - single locations, 8
- Executing a machine language program
  - from Applesoft, 192–193
  - from the monitor, 13, 29
- Expanding a number, 2, 191, 195



FLASH, 41-44, 225-227

GET, 35-36

Go, 13, 29, 199

GOSUB, 26, 193

GOTO, 73

#### Graphics

entering graphics mode, 27

high resolution graphics (shape tables),  
153-171

plotting horizontal lines, 29-30

plotting points, 26-27

plotting vertical lines, 30-31

HCOLOR, 165

Hexadecimal (HEX or base 16), 1

addition using the monitor, 50

conversion to binary, 3-4

conversion to decimal, 4 (problem), 191,  
195

subtraction using the monitor, 51

HGR, 165

High resolution graphics (shape tables),  
153-171

HIMEM:, 197

HLIN, 29-30

HOME, 47

IF statement, 56

branch command-compare command,  
equivalents of, 65-73

Immediate addressing mode, 15

Implied addressing mode, 15, 66

INC, 67

operation in decimal mode, 129

Index (of a shape table), 153, 161-162

Indexed indirect addressing mode, 179

Indirect addressing modes, 175-190

indexed indirect addressing mode, 179

indirect addressing mode, 175-178

indirect indexed addressing mode, 179

postindexed indirect addressing mode,  
179, 181-183

preindexed indirect addressing mode,  
178-181

Indirect indexed addressing mode, 179

Input of values from the keyboard, 35-36

input of two-digit numbers from the  
keyboard, 40, 129-134

use of AND, 39-40

Integration of machine language programs  
with Applesoft programs, 190-197

Interchange sort algorithm, 100-107

Intrinsic subroutines. *See* Built-in sub-  
routines

INVERSE, 41-44, 225-227

INX, 67

operation in decimal mode, 129

INY, 67

operation in decimal mode, 129

JMP, 73-75

absolute addressing mode, 73, 176

indirect addressing mode, 175-178

JSR, 26, 75-76

L command (monitor), 16

Labeling output, 33-34

LDA, 11-12, 15

absolute indexed addressing mode, 45-46

zero page addressing mode, 28

LDA SC030 (speaker toggle command), 112

LDX, 27, 28

zero page addressing mode, 28

LDY, 17, 27

Least significant byte (LSB), 11, 28

Loops, 56

examples of, 66-69

Machine language subroutines in Applesoft  
programs, 190-197

#### Matrices

addition, 201 (problem)

programming examples, 183-190

Median, 201 (problem)

#### Memory

dump, 8

for storing machine language programs,  
6-7

for storing shape tables, 162-163

locations, 2, 4

random access (RAM), 6-7

ranges, 8

read only (ROM), 6-7



- Menu program, 175
- Mini-assembler, 11, 28, 92–97
  - branch commands, 95
  - entering the mini-assembler program, 93
  - leaving the mini-assembler program, 96
  - monitor commands from the mini-assembler program, 96–97
  - notations for addressing modes, 94, 177, 181, 183
- Mnemonic codes, 11
  - list of, 206–217
- Monitor commands
  - changing memory, 9–10
  - colon, 9
  - examining memory, 7–9
  - Go command, 13, 29, 199
  - List command, 16
  - Move command, 18
  - Step command, 197–198
  - Trace command, 198
- Monitor program, 4, 6
  - changing the contents of memory, 9–10
  - entering and leaving the monitor, 7
  - entering shape tables, 162–163
  - examining memory, 7–9
  - format of display, 9
- Most significant byte (MSB), 11, 27
- Motion graphics, 170
- Move command (monitor), 18
- Moving data, 10
- Multiplication, 140–150
  
- N bit (processor status register), 56–57
- Negative numbers, 51–52
  - branch command operands, 52
- Nested loops
  - interchange sort algorithm, 100–107
  - miscellaneous examples, 88–92, 97–100
  - nesting more than two loops, 107–111
  - production of sound, 111–117
- Nibble, 4
- NORMAL, 41–44, 225–227
- Number systems, 1–5, *see also* Converting
  - from one base to another, Decimal number system, Hexadecimal
- Op codes
  - definition of, 11
  - list of, 206–217
- Operands 11–12
  
- P register. *See* Processor status register
- PEEK, 192
- Place value, 1, 130
- PLOT, 26–27
- Plotting points, 26–27
- Plotting vectors, 153–158
- POKE, 191–192
  - machine language equivalent, 29
- Postindexed indirect addressing mode, 179, 181–183
  - assembly language notation, 182, 183
  - programming example, 184–187
- PR #. 6. 55
- Preindexed indirect addressing mode, 178–181
  - assembly language notation, 180, 181
  - programming example, 187–190
- Printing characters, 33–34
  - memory locations controlling the text screen, 45, 228–229
- NORMAL, INVERSE and FLASH
  - modes, 41–44, 225–227
- Printing numerical information, 34–35
- Problems, 4–5, 22–24, 48–49, 84–87, 117–120, 150–152, 171–174, 199–202
- Processor status register, 56–57, 206
  - C bit, 122, 126
  - diagram of register, 206
  - N bit, 56–57
  - Z bit, 56–57
  
- Random access memory (RAM), 6–7
- Random numbers, 36–37
  - example, 105–107
- Read only memory (ROM), 6–7
- Relative addressing mode, 60
  - assembly language notation, 65
  - branch commands, 59–65
  - calculation of operands, 60–65
  - operands, 59–65
- RETURN, 193
- ROL, 146–148
- ROR, 148–149
- ROT, 163–164
- RTS, 12, 15, 76
  - in debugging, 199
  - in subroutines, 76



- S command (monitor), 197–198
- Saving machine language programs, 54–55
- Saving shape tables, 163
- SBC, 138
- SCALE, 163–164
- SEC, 138
- SED, 128
- Shape definition, 153, 155–160
- Shape tables, 153–171
  - computer generated “art”, 171
  - DRAW, 163, 165
  - entering shape tables into memory, 162–163
  - erasing shapes, 166–167
  - HCOLOR, 165
  - HGR, 165
  - index, 153, 161–162
  - memory locations for, 162–163
  - motion graphics, 170
  - plotting vectors, 153–158
  - ROT, 163–164
  - saving shape tables on disk, 163
  - SCALE, 163, 164
  - schematic diagram of shape table, 161
  - shape definition, 153, 155–160
  - shape table construction, 160–162, 167–170
  - using shape tables, 163–167
  - XDRAW, 163, 166–167
- Sort algorithms
  - bubble sort, 77–84
  - interchange method, 100–107
- Sounds
  - duration of notes, 112, 113–115
  - pitch of notes, 111–113
  - programming example, 115–117
  - speaker toggle, 111
- Speaker, 111
- STA, 11, 12
  - absolute indexed addressing mode, 46
  - zero page addressing mode, 28
- Step command (monitor), 197–198
- Subroutines, 75–76
  - JSR, 26, 75–76
  - list of, 220–223
  - RTS, 12, 15, 76
  - see also* Built-in subroutines
- Subscripted variables, 45–46
- Subtraction
  - examples, 52–54
  - program, 138–140
  - using the monitor, 51
- T command (monitor), 198
- TAX, 40
- TAY, 39, 40
- TEXT, 29
- Text screen, 45, 183–184
  - programming examples, 183–190
  - text screen memory map, 228–229
- Trace command, 198
- TXA, 40
- TYA, 40
- VLIN, 30–31
- Writing machine language programs, 13–14
- X register, 25
  - absolute indexed addressing mode, 45–46
  - effect of built-in subroutines on, 31–32, 220–223
  - preindexed indirect addressing mode, 179
- XDRAW, 163
  - erasing shapes, 166–167
- Y register, 25
  - absolute indexed addressing mode, 45–46
  - effect of built-in subroutines on, 31–32, 137, 220–223
  - postindexed indirect addressing mode, 181–183
- Z bit, 56–57
- Zero page addressing mode, 28
- Zero page indexed addressing mode, 179
- Zero page memory, 27–28
  - postindexed indirect addressing mode, 181–183
  - preindexed indirect addressing mode, 178–181













ISBN 0-03-063336-2